Groups

Release 10.6

The Sage Development Team

Apr 01, 2025

CONTENTS

CHAPTER ONE

EXAMPLES OF GROUPS

The `groups` object may be used to access examples of various groups. Using tab-completion on this object is an easy way to discover and quickly create the groups that are available (as listed here).

Let `<tab>` indicate pressing the `Tab` key. So begin by typing `groups.<tab>` to the see primary divisions, followed by (for example) `groups.matrix.<tab>` to access various groups implemented as sets of matrices.

- Permutation Groups (`groups.permutation.<tab>`)
  - *groups.permutation.Symmetric*
  - *groups.permutation.Alternating*
  - *groups.permutation.KleinFour*
  - *groups.permutation.Quaternion*
  - *groups.permutation.Cyclic*
  - *groups.permutation.ComplexReflection*
  - *groups.permutation.Dihedral*
  - *groups.permutation.DiCyclic*
  - *groups.permutation.Mathieu*
  - *groups.permutation.Suzuki*
  - *groups.permutation.PGL*
  - *groups.permutation.PSL*
  - *groups.permutation.PSp*
  - *groups.permutation.PSU*
  - *groups.permutation.PGU*
  - *groups.permutation.Transitive*
  - *groups.permutation.RubiksCube*
- Matrix Groups (`groups.matrix.<tab>`)
  - *groups.matrix.QuaternionGF3*
  - *groups.matrix.GL*
  - *groups.matrix.SL*
  - *groups.matrix.Sp*
  - *groups.matrix.GU*

  - *groups.matrix.SU*
  - *groups.matrix.GO*
  - *groups.matrix.SO*
- Finitely Presented Groups (`groups.presentation.<tab>`)
  - *groups.presentation.Alternating*
  - *groups.presentation.Cactus*
  - *groups.presentation.Cyclic*
  - *groups.presentation.Dihedral*
  - *groups.presentation.DiCyclic*
  - *groups.presentation.FGAbelian*
  - *groups.presentation.KleinFour*
  - *groups.presentation.Quaternion*
  - *groups.presentation.Symmetric*
- Affine Groups (`groups.affine.<tab>`)
  - *groups.affine.Affine*
  - *groups.affine.Euclidean*
- Lie Groups (`groups.lie.<tab>`)
  - *groups.lie.Nilpotent*
- Miscellaneous Groups (`groups.misc.<tab>`)
  - Coxeter, reflection and related groups
    * *groups.misc.Braid*
    * *groups.misc.Cactus*
    * `groups.misc.CoxeterGroup`
    * *groups.misc.PureCactus*
    * `groups.misc.ReflectionGroup`
    * *groups.misc.RightAngledArtin*
    * `groups.misc.WeylGroup`
  - other miscellaneous groups
    * *groups.misc.AdditiveAbelian*
    * `groups.misc.AdditiveCyclic`
    * *groups.misc.Free*
    * *groups.misc.SemimonomialTransformation*

CHAPTER

TWO

BASE CLASS FOR GROUPS

class sage.groups.group.**AbelianGroup**

Bases: *Group*

Generic abelian group.

is_abelian()

Return True.

EXAMPLES:

```
sage: from sage.groups.group import AbelianGroup
sage: G = AbelianGroup()
sage: G.is_abelian()
True
```

```
>>> from sage.all import *
>>> from sage.groups.group import AbelianGroup
>>> G = AbelianGroup()
>>> G.is_abelian()
True
```

class sage.groups.group.**AlgebraicGroup**

Bases: *Group*

class sage.groups.group.**FiniteGroup**

Bases: *Group*

Generic finite group.

is_finite()

Return True.

EXAMPLES:

```
sage: from sage.groups.group import FiniteGroup
sage: G = FiniteGroup()
sage: G.is_finite()
True
```

```
>>> from sage.all import *
>>> from sage.groups.group import FiniteGroup
>>> G = FiniteGroup()
```

(continues on next page)

(continued from previous page)

```
>>> G.is_finite()
True
```

class sage.groups.group.**Group**

Bases: Parent

Base class for all groups.

is_abelian()

Test whether this group is abelian.

EXAMPLES:

```
sage: from sage.groups.group import Group
sage: G = Group()
sage: G.is_abelian()
Traceback (most recent call last):
...
NotImplementedError
```

```
>>> from sage.all import *
>>> from sage.groups.group import Group
>>> G = Group()
>>> G.is_abelian()
Traceback (most recent call last):
...
NotImplementedError
```

is_commutative()

Test whether this group is commutative.

This is an alias for is_abelian, largely to make groups work well with the Factorization class.

(Note for developers: Derived classes should override is_abelian, not is_commutative.)

EXAMPLES:

```
sage: SL(2, 7).is_commutative()                                      #␣
↪needs sage.libs.gap sage.modules sage.rings.finite_rings
False
```

```
>>> from sage.all import *
>>> SL(Integer(2), Integer(7)).is_commutative()                      ␣
↪               # needs sage.libs.gap sage.modules sage.rings.finite_rings
False
```

is_finite()

Return `True` if this group is finite.

EXAMPLES:

```
sage: from sage.groups.group import Group
sage: G = Group()
sage: G.is_finite()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
NotImplementedError
```

```
>>> from sage.all import *
>>> from sage.groups.group import Group
>>> G = Group()
>>> G.is_finite()
Traceback (most recent call last):
...
NotImplementedError
```

is_multiplicative()

Return `True` if the group operation is given by `*` (rather than `+`).

Override for additive groups.

EXAMPLES:

```
sage: from sage.groups.group import Group
sage: G = Group()
sage: G.is_multiplicative()
True
```

```
>>> from sage.all import *
>>> from sage.groups.group import Group
>>> G = Group()
>>> G.is_multiplicative()
True
```

is_trivial()

Return `True` if this group is the trivial group.

A group is trivial, if it consists only of the identity element.

> **Warning**
>
> It is in principle undecidable whether a group is trivial, for example, if the group is given by a finite presentation. Thus, this method may not terminate.

EXAMPLES:

```
sage: groups.presentation.Cyclic(1).is_trivial()
True

sage: G.<a,b> = FreeGroup('a, b')
sage: H = G / (a^2, b^3, a*b*~a*~b)
sage: H.is_trivial()
False
```

```
>>> from sage.all import *
>>> groups.presentation.Cyclic(Integer(1)).is_trivial()
```

(continues on next page)

(continued from previous page)

```
True

>>> G = FreeGroup('a, b', names=('a', 'b',)); (a, b,) = G._first_ngens(2)
>>> H = G / (a**Integer(2), b**Integer(3), a*b*~a*~b)
>>> H.is_trivial()
False
```

A non-trivial presentation of the trivial group:

```
sage: F.<a,b> = FreeGroup()
sage: J = F / ((~a)*b*a*(~b)^2, (~b)*a*b*(~a)^2)
sage: J.is_trivial()
True
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('a', 'b',)); (a, b,) = F._first_ngens(2)
>>> J = F / ((~a)*b*a*(~b)**Integer(2), (~b)*a*b*(~a)**Integer(2))
>>> J.is_trivial()
True
```

order()

Return the number of elements of this group.

This is either a positive integer or infinity.

EXAMPLES:

```
sage: from sage.groups.group import Group
sage: G = Group()
sage: G.order()
Traceback (most recent call last):
...
NotImplementedError
```

```
>>> from sage.all import *
>>> from sage.groups.group import Group
>>> G = Group()
>>> G.order()
Traceback (most recent call last):
...
NotImplementedError
```

quotient(*H*, ***kwds*)

Return the quotient of this group by the normal subgroup H.

EXAMPLES:

```
sage: from sage.groups.group import Group
sage: G = Group()
sage: G.quotient(G)
Traceback (most recent call last):
...
NotImplementedError
```

```
>>> from sage.all import *
>>> from sage.groups.group import Group
>>> G = Group()
>>> G.quotient(G)
Traceback (most recent call last):
...
NotImplementedError
```

sage.groups.group.**is_Group**(*x*)

Return whether x is a group object.

INPUT:

- x – anything

OUTPUT: boolean

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()                                                    #␣
↪needs sage.groups
sage: from sage.groups.group import is_Group
sage: is_Group(F)                                                              #␣
↪needs sage.groups
doctest:warning...DeprecationWarning: use instead G in Groups()
See https://github.com/sagemath/sage/issues/37449 for details.
True
sage: is_Group("a string")
False
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('a', 'b',)); (a, b,) = F._first_ngens(2)# needs sage.
↪groups
>>> from sage.groups.group import is_Group
>>> is_Group(F)                                                               #␣
↪needs sage.groups
doctest:warning...DeprecationWarning: use instead G in Groups()
See https://github.com/sagemath/sage/issues/37449 for details.
True
>>> is_Group("a string")
False
```

CHAPTER

THREE

GROUP HOMOMORPHISMS FOR GROUPS WITH A GAP BACKEND

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: A = AbelianGroupGap([2, 4])
sage: F.<a,b> = FreeGroup()
sage: f = F.hom([g for g in A.gens()])
sage: K = f.kernel()
sage: K
Group(<free, no generators known>)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> A = AbelianGroupGap([Integer(2), Integer(4)])
>>> F = FreeGroup(names=('a', 'b',)); (a, b,) = F._first_ngens(2)
>>> f = F.hom([g for g in A.gens()])
>>> K = f.kernel()
>>> K
Group(<free, no generators known>)
```

AUTHORS:

- Simon Brandhorst (2018-02-08): initial version
- Sebastian Oehms (2018-11-15): have this functionality work for permutation groups (Issue #26750) and implement `section()` and `natural_map()`

class `sage.groups.libgap_morphism.`**`GroupHomset_libgap`**(*G*, *H*, *category=None*, *check=True*)

Bases: `HomsetWithBase`

Homsets of groups with a libgap backend.

Do not call this directly instead use `Hom()`.

INPUT:

- `G` – a libgap group
- `H` – a libgap group
- `category` – a category

OUTPUT: the homset of two libgap groups

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: A = AbelianGroupGap([2,4])
sage: H = A.Hom(A)
sage: H
Set of Morphisms from Abelian group with gap, generator orders (2, 4)
 to Abelian group with gap, generator orders (2, 4)
 in Category of finite enumerated commutative groups
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> A = AbelianGroupGap([Integer(2),Integer(4)])
>>> H = A.Hom(A)
>>> H
Set of Morphisms from Abelian group with gap, generator orders (2, 4)
 to Abelian group with gap, generator orders (2, 4)
 in Category of finite enumerated commutative groups
```

Element

alias of *GroupMorphism_libgap*

natural_map()

This method from `HomsetWithBase` is overloaded here for cases in which both groups have corresponding lists of generators.

OUTPUT:

An instance of the element class of `self` if there exists a group homomorphism mapping the generators of the domain of `self` to the according generators of the codomain. Otherwise, the method falls back to the default.

EXAMPLES:

```
sage: G = GL(3,2)
sage: P = PGL(3,2)
sage: nat = Hom(G, P).natural_map()
sage: type(nat)
<class 'sage.groups.libgap_morphism.GroupHomset_libgap_with_category.element_
→class'>
sage: g1, g2 = G.gens()
sage: nat(g1*g2)
(1,2,4,5,7,3,6)
```

```
>>> from sage.all import *
>>> G = GL(Integer(3),Integer(2))
>>> P = PGL(Integer(3),Integer(2))
>>> nat = Hom(G, P).natural_map()
>>> type(nat)
<class 'sage.groups.libgap_morphism.GroupHomset_libgap_with_category.element_
→class'>
>>> g1, g2 = G.gens()
>>> nat(g1*g2)
(1,2,4,5,7,3,6)
```

class sage.groups.libgap_morphism.**GroupMorphism_libgap**(*homset*, *gap_hom*, *check=True*)

Bases: `Morphism`

This wraps GAP group homomorphisms.

Checking if the input defines a group homomorphism can be expensive if the group is large.

INPUT:

- `homset` – the parent
- `gap_hom` – a `sage.libs.gap.element.GapElement` consisting of a group homomorphism
- `check` – boolean (default: `True`); check if the `gap_hom` is a group homomorphism (this can be expensive)

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: A = AbelianGroupGap([2, 4])
sage: A.hom([g^2 for g in A.gens()])
Group endomorphism of Abelian group with gap, generator orders (2, 4)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> A = AbelianGroupGap([Integer(2), Integer(4)])
>>> A.hom([g**Integer(2) for g in A.gens()])
Group endomorphism of Abelian group with gap, generator orders (2, 4)
```

Homomorphisms can be defined between different kinds of GAP groups:

```
sage: G = MatrixGroup([Matrix(ZZ, 2, [0,1,1,0])])
sage: f = A.hom([G.0, G(1)])
sage: f
Group morphism:
From: Abelian group with gap, generator orders (2, 4)
To:   Matrix group over Integer Ring with 1 generators (
[0 1]
[1 0]
)
sage: G.<a,b> = FreeGroup()
sage: H = G / (G([1]), G([2])^3)
sage: f = G.hom(H.gens())
sage: f
Group morphism:
  From: Free Group on generators {a, b}
  To:   Finitely presented group < a, b | a, b^3 >
```

```
>>> from sage.all import *
>>> G = MatrixGroup([Matrix(ZZ, Integer(2), [Integer(0),Integer(1),Integer(1),
↪Integer(0)])])
>>> f = A.hom([G.gen(0), G(Integer(1))])
>>> f
Group morphism:
From: Abelian group with gap, generator orders (2, 4)
To:   Matrix group over Integer Ring with 1 generators (
[0 1]
[1 0]
)
>>> G = FreeGroup(names=('a', 'b',)); (a, b,) = G._first_ngens(2)
```

(continues on next page)

(continued from previous page)

```
>>> H = G / (G([Integer(1)]), G([Integer(2)])**Integer(3))
>>> f = G.hom(H.gens())
>>> f
Group morphism:
  From: Free Group on generators {a, b}
  To:   Finitely presented group < a, b | a, b^3 >
```

Homomorphisms can be defined between GAP groups and permutation groups:

```
sage: S = Sp(4,3)
sage: P = PSp(4,3)
sage: pr = S.hom(P.gens())
sage: E = copy(S.one().matrix())
sage: E[3,0] = 2; e = S(E)
sage: pr(e)
(1,16,15)(3,22,18)(4,19,21)(6,34,24)(7,25,33)(9,40,27)(10,28,39)(12,37,30)(13,31,
↪36)
```

```
>>> from sage.all import *
>>> S = Sp(Integer(4),Integer(3))
>>> P = PSp(Integer(4),Integer(3))
>>> pr = S.hom(P.gens())
>>> E = copy(S.one().matrix())
>>> E[Integer(3),Integer(0)] = Integer(2); e = S(E)
>>> pr(e)
(1,16,15)(3,22,18)(4,19,21)(6,34,24)(7,25,33)(9,40,27)(10,28,39)(12,37,30)(13,31,
↪36)
```

gap()

Return the underlying LibGAP group homomorphism.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: A = AbelianGroupGap([2,4])
sage: f = A.hom([g^2 for g in A.gens()])
sage: f.gap()
[ f1, f2 ] -> [ <identity> of ..., f3 ]
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> A = AbelianGroupGap([Integer(2),Integer(4)])
>>> f = A.hom([g**Integer(2) for g in A.gens()])
>>> f.gap()
[ f1, f2 ] -> [ <identity> of ..., f3 ]
```

image(*J*, **args*, ***kwds*)

The image of an element or a subgroup.

INPUT:

- `J` – a subgroup or an element of the domain of `self`

OUTPUT: the image of `J` under `self`

> **Note**
>
> `pushforward` is the method that is used when a map is called on anything that is not an element of its domain. For historical reasons, we keep the alias `image()` for this method.

EXAMPLES:

```
sage: G.<a,b> = FreeGroup()
sage: H = G / (G([1]), G([2])^3)
sage: f = G.hom(H.gens())
sage: S = G.subgroup([a.gap()])
sage: f.pushforward(S)
Group([ a ])
sage: x = f.image(a)
sage: x
a
sage: x.parent()
Finitely presented group < a, b | a, b^3 >

sage: # needs sage.rings.finite_rings
sage: G = GU(3,2)
sage: P = PGU(3,2)
sage: pr = Hom(G, P).natural_map()
sage: GS = G.subgroup([G.gen(0)])
sage: pr.pushforward(GS)
Subgroup generated by [(3,4,5)(10,18,14)(11,19,15)(12,20,16)(13,21,17)] of
(The projective general unitary group of degree 3 over Finite Field of size 2)
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('a', 'b',)); (a, b,) = G._first_ngens(2)
>>> H = G / (G([Integer(1)]), G([Integer(2)])**Integer(3))
>>> f = G.hom(H.gens())
>>> S = G.subgroup([a.gap()])
>>> f.pushforward(S)
Group([ a ])
>>> x = f.image(a)
>>> x
a
>>> x.parent()
Finitely presented group < a, b | a, b^3 >

>>> # needs sage.rings.finite_rings
>>> G = GU(Integer(3),Integer(2))
>>> P = PGU(Integer(3),Integer(2))
>>> pr = Hom(G, P).natural_map()
>>> GS = G.subgroup([G.gen(Integer(0))])
>>> pr.pushforward(GS)
Subgroup generated by [(3,4,5)(10,18,14)(11,19,15)(12,20,16)(13,21,17)] of
(The projective general unitary group of degree 3 over Finite Field of size 2)
```

kernel()

Return the kernel of `self`.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: A1 = AbelianGroupGap([6, 6])
sage: A2 = AbelianGroupGap([3, 3])
sage: f = A1.hom(A2.gens())
sage: f.kernel()
Subgroup of Abelian group with gap, generator orders (6, 6)
 generated by (f1*f2, f3*f4)
sage: f.kernel().order()
4
sage: S = Sp(6,3)
sage: P = PSp(6,3)
sage: pr = Hom(S, P).natural_map()
sage: pr.kernel()
Subgroup with 1 generators (
[2 0 0 0 0 0]
[0 2 0 0 0 0]
[0 0 2 0 0 0]
[0 0 0 2 0 0]
[0 0 0 0 2 0]
[0 0 0 0 0 2]
) of Symplectic Group of degree 6 over Finite Field of size 3
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> A1 = AbelianGroupGap([Integer(6), Integer(6)])
>>> A2 = AbelianGroupGap([Integer(3), Integer(3)])
>>> f = A1.hom(A2.gens())
>>> f.kernel()
Subgroup of Abelian group with gap, generator orders (6, 6)
 generated by (f1*f2, f3*f4)
>>> f.kernel().order()
4
>>> S = Sp(Integer(6),Integer(3))
>>> P = PSp(Integer(6),Integer(3))
>>> pr = Hom(S, P).natural_map()
>>> pr.kernel()
Subgroup with 1 generators (
[2 0 0 0 0 0]
[0 2 0 0 0 0]
[0 0 2 0 0 0]
[0 0 0 2 0 0]
[0 0 0 0 2 0]
[0 0 0 0 0 2]
) of Symplectic Group of degree 6 over Finite Field of size 3
```

lift(*h*)

Return an element of the domain that maps to `h`.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: A = AbelianGroupGap([2,4])
```

(continues on next page)

(continued from previous page)

```
sage: f = A.hom([g^2 for g in A.gens()])
sage: a = A.gens()[1]
sage: f.lift(a^2)
f2
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> A = AbelianGroupGap([Integer(2),Integer(4)])
>>> f = A.hom([g**Integer(2) for g in A.gens()])
>>> a = A.gens()[Integer(1)]
>>> f.lift(a**Integer(2))
f2
```

If the element is not in the image, we raise an error:

```
sage: f.lift(a)
Traceback (most recent call last):
...
ValueError: f2 is not an element of the image of Group endomorphism
 of Abelian group with gap, generator orders (2, 4)
```

```
>>> from sage.all import *
>>> f.lift(a)
Traceback (most recent call last):
...
ValueError: f2 is not an element of the image of Group endomorphism
 of Abelian group with gap, generator orders (2, 4)
```

preimage(S)

Return the preimage of the subgroup `S`.

INPUT:

- `S` – a subgroup of this group

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: A = AbelianGroupGap([2,4])
sage: B = AbelianGroupGap([4])
sage: f = A.hom([B.one(), B.gen(0)^2])
sage: S = B.subgroup([B.one()])
sage: f.preimage(S) == f.kernel()
True
sage: S = Sp(4,3)
sage: P = PSp(4,3)
sage: pr = Hom(S, P).natural_map()
sage: PS = P.subgroup([P.gen(0)])
sage: pr.preimage(PS)
Subgroup with 2 generators (
[2 0 0 0]  [1 0 0 0]
[0 2 0 0]  [0 2 0 0]
[0 0 2 0]  [0 0 2 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 0 2], [0 0 0 1]
) of Symplectic Group of degree 4 over Finite Field of size 3
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> A = AbelianGroupGap([Integer(2),Integer(4)])
>>> B = AbelianGroupGap([Integer(4)])
>>> f = A.hom([B.one(), B.gen(Integer(0))**Integer(2)])
>>> S = B.subgroup([B.one()])
>>> f.preimage(S) == f.kernel()
True
>>> S = Sp(Integer(4),Integer(3))
>>> P = PSp(Integer(4),Integer(3))
>>> pr = Hom(S, P).natural_map()
>>> PS = P.subgroup([P.gen(Integer(0))])
>>> pr.preimage(PS)
Subgroup with 2 generators (
[2 0 0 0]  [1 0 0 0]
[0 2 0 0]  [0 2 0 0]
[0 0 2 0]  [0 0 2 0]
[0 0 0 2], [0 0 0 1]
) of Symplectic Group of degree 4 over Finite Field of size 3
```

pushforward(*J*, **args*, ***kwds*)

The image of an element or a subgroup.

INPUT:

- `J` – a subgroup or an element of the domain of `self`

OUTPUT: the image of `J` under `self`

> **Note**
>
> `pushforward` is the method that is used when a map is called on anything that is not an element of its domain. For historical reasons, we keep the alias `image()` for this method.

EXAMPLES:

```
sage: G.<a,b> = FreeGroup()
sage: H = G / (G([1]), G([2])^3)
sage: f = G.hom(H.gens())
sage: S = G.subgroup([a.gap()])
sage: f.pushforward(S)
Group([ a ])
sage: x = f.image(a)
sage: x
a
sage: x.parent()
Finitely presented group < a, b | a, b^3 >

sage: # needs sage.rings.finite_rings
```

(continues on next page)

(continued from previous page)

```
sage: G = GU(3,2)
sage: P = PGU(3,2)
sage: pr = Hom(G, P).natural_map()
sage: GS = G.subgroup([G.gen(0)])
sage: pr.pushforward(GS)
Subgroup generated by [(3,4,5)(10,18,14)(11,19,15)(12,20,16)(13,21,17)] of
(The projective general unitary group of degree 3 over Finite Field of size 2)
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('a', 'b',)); (a, b,) = G._first_ngens(2)
>>> H = G / (G([Integer(1)]), G([Integer(2)])**Integer(3))
>>> f = G.hom(H.gens())
>>> S = G.subgroup([a.gap()])
>>> f.pushforward(S)
Group([ a ])
>>> x = f.image(a)
>>> x
a
>>> x.parent()
Finitely presented group < a, b | a, b^3 >

>>> # needs sage.rings.finite_rings
>>> G = GU(Integer(3),Integer(2))
>>> P = PGU(Integer(3),Integer(2))
>>> pr = Hom(G, P).natural_map()
>>> GS = G.subgroup([G.gen(Integer(0))])
>>> pr.pushforward(GS)
Subgroup generated by [(3,4,5)(10,18,14)(11,19,15)(12,20,16)(13,21,17)] of
(The projective general unitary group of degree 3 over Finite Field of size 2)
```

section()

Return a section map of `self` by use of *lift()*.

See *section()* of `sage.categories.map.Map`, as well.

OUTPUT: an instance of `sage.categories.morphism.SetMorphism` mapping an element of the codomain of `self` to one of its preimages

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: G = GU(3,2)
sage: P = PGU(3,2)
sage: pr = Hom(G, P).natural_map()
sage: sect = pr.section()
sage: sect(P.an_element())
[a + 1     a     a]
[    1     1     0]
[    a     0     0]
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> G = GU(Integer(3),Integer(2))
```

(continues on next page)

(continued from previous page)

```
>>> P = PGU(Integer(3),Integer(2))
>>> pr = Hom(G, P).natural_map()
>>> sect = pr.section()
>>> sect(P.an_element())
[a + 1     a     a]
[    1     1     0]
[    a     0     0]
```

CHAPTER

FOUR

LIBGAP-BASED GROUPS

This module provides helper class for wrapping GAP groups via `libgap`. See *free_group* for an example how they are used.

The parent class keeps track of the GAP element object, to use it in your Python parent you have to derive both from the suitable group parent and *ParentLibGAP*

```
sage: from sage.groups.libgap_wrapper import ElementLibGAP, ParentLibGAP
sage: from sage.groups.group import Group
sage: class FooElement(ElementLibGAP):
....:     pass
sage: class FooGroup(Group, ParentLibGAP):
....:     Element = FooElement
....:     def __init__(self):
....:         lg = libgap(libgap.CyclicGroup(3))     # dummy
....:         ParentLibGAP.__init__(self, lg)
....:         Group.__init__(self)
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_wrapper import ElementLibGAP, ParentLibGAP
>>> from sage.groups.group import Group
>>> class FooElement(ElementLibGAP):
...     pass
>>> class FooGroup(Group, ParentLibGAP):
...     Element = FooElement
...     def __init__(self):
...         lg = libgap(libgap.CyclicGroup(Integer(3)))     # dummy
...         ParentLibGAP.__init__(self, lg)
...         Group.__init__(self)
```

Note how we call the constructor of both superclasses to initialize `Group` and `ParentLibGAP` separately. The parent class implements its output via LibGAP:

```
sage: FooGroup()
<pc group of size 3 with 1 generator>
sage: type(FooGroup().gap())
<class 'sage.libs.gap.element.GapElement'>
```

```
>>> from sage.all import *
>>> FooGroup()
<pc group of size 3 with 1 generator>
```

(continues on next page)

(continued from previous page)

```
>>> type(FooGroup().gap())
<class 'sage.libs.gap.element.GapElement'>
```

The element class is a subclass of `MultiplicativeGroupElement`. To use it, you just inherit from *`ElementLibGAP`*

```
sage: element = FooGroup().an_element()
sage: element
f1
```

```
>>> from sage.all import *
>>> element = FooGroup().an_element()
>>> element
f1
```

The element class implements group operations and printing via LibGAP:

```
sage: element._repr_()
'f1'
sage: element * element
f1^2
```

```
>>> from sage.all import *
>>> element._repr_()
'f1'
>>> element * element
f1^2
```

AUTHORS:

- Volker Braun

class `sage.groups.libgap_wrapper.`**`ElementLibGAP`**

Bases: `MultiplicativeGroupElement`

A class for LibGAP-based Sage group elements.

INPUT:

- `parent` – the Sage parent
- `libgap_element` – the libgap element that is being wrapped

EXAMPLES:

```
sage: from sage.groups.libgap_wrapper import ElementLibGAP, ParentLibGAP
sage: from sage.groups.group import Group
sage: class FooElement(ElementLibGAP):
....:     pass
sage: class FooGroup(Group, ParentLibGAP):
....:     Element = FooElement
....:     def __init__(self):
....:         lg = libgap(libgap.CyclicGroup(3))    # dummy
....:         ParentLibGAP.__init__(self, lg)
....:         Group.__init__(self)
sage: FooGroup()
```

(continues on next page)

(continued from previous page)

```
<pc group of size 3 with 1 generator>
sage: FooGroup().gens()
(f1,)
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_wrapper import ElementLibGAP, ParentLibGAP
>>> from sage.groups.group import Group
>>> class FooElement(ElementLibGAP):
...     pass
>>> class FooGroup(Group, ParentLibGAP):
...     Element = FooElement
...     def __init__(self):
...         lg = libgap(libgap.CyclicGroup(Integer(3)))     # dummy
...         ParentLibGAP.__init__(self, lg)
...         Group.__init__(self)
>>> FooGroup()
<pc group of size 3 with 1 generator>
>>> FooGroup().gens()
(f1,)
```

gap()

Return a LibGAP representation of the element.

OUTPUT: a `GapElement`

EXAMPLES:

```
sage: G.<a,b> = FreeGroup('a, b')
sage: x = G([1, 2, -1, -2])
sage: x
a*b*a^-1*b^-1
sage: xg = x.gap()
sage: xg
a*b*a^-1*b^-1
sage: type(xg)
<class 'sage.libs.gap.element.GapElement'>
```

```
>>> from sage.all import *
>>> G = FreeGroup('a, b', names=('a', 'b',)); (a, b,) = G._first_ngens(2)
>>> x = G([Integer(1), Integer(2), -Integer(1), -Integer(2)])
>>> x
a*b*a^-1*b^-1
>>> xg = x.gap()
>>> xg
a*b*a^-1*b^-1
>>> type(xg)
<class 'sage.libs.gap.element.GapElement'>
```

inverse()

Return the inverse of `self`.

is_conjugate(*other*)

Return whether the elements `self` and `other` are conjugate.

EXAMPLES:

```
sage: from sage.groups.libgap_group import GroupLibGAP
sage: G = GroupLibGAP(libgap.GL(2, 3))
sage: a,b = G.gens()
sage: a.is_conjugate(b)
False
sage: a.is_conjugate((a*b^2) * a * ~(a*b^2))
True
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_group import GroupLibGAP
>>> G = GroupLibGAP(libgap.GL(Integer(2), Integer(3)))
>>> a,b = G.gens()
>>> a.is_conjugate(b)
False
>>> a.is_conjugate((a*b**Integer(2)) * a * ~(a*b**Integer(2)))
True
```

is_one()

Test whether the group element is the trivial element.

OUTPUT: boolean

EXAMPLES:

```
sage: G.<a,b> = FreeGroup('a, b')
sage: x = G([1, 2, -1, -2])
sage: x.is_one()
False
sage: (x * ~x).is_one()
True
```

```
>>> from sage.all import *
>>> G = FreeGroup('a, b', names=('a', 'b',)); (a, b,) = G._first_ngens(2)
>>> x = G([Integer(1), Integer(2), -Integer(1), -Integer(2)])
>>> x.is_one()
False
>>> (x * ~x).is_one()
True
```

multiplicative_order()

Return the multiplicative order.

EXAMPLES:

```
sage: from sage.groups.libgap_group import GroupLibGAP
sage: G = GroupLibGAP(libgap.GL(2, 3))
sage: a,b = G.gens()
sage: print(a.order())
2
sage: print(a.multiplicative_order())
2
```

(continues on next page)

(continued from previous page)

```
sage: z = Mod(0, 3)
sage: o = Mod(1, 3)
sage: G(libgap([[o,o],[z,o]])).order()
3
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_group import GroupLibGAP
>>> G = GroupLibGAP(libgap.GL(Integer(2), Integer(3)))
>>> a,b = G.gens()
>>> print(a.order())
2
>>> print(a.multiplicative_order())
2

>>> z = Mod(Integer(0), Integer(3))
>>> o = Mod(Integer(1), Integer(3))
>>> G(libgap([[o,o],[z,o]])).order()
3
```

normalizer()

Return the normalizer of the cyclic group generated by this element.

EXAMPLES:

```
sage: from sage.groups.libgap_group import GroupLibGAP
sage: G = GroupLibGAP(libgap.GL(3,3))
sage: a,b = G.gens()
sage: H = a.normalizer()
sage: H
<group of 3x3 matrices over GF(3)>
sage: H.cardinality()
96
sage: all(g*a == a*g for g in H)
True
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_group import GroupLibGAP
>>> G = GroupLibGAP(libgap.GL(Integer(3),Integer(3)))
>>> a,b = G.gens()
>>> H = a.normalizer()
>>> H
<group of 3x3 matrices over GF(3)>
>>> H.cardinality()
96
>>> all(g*a == a*g for g in H)
True
```

nth_roots(*n*)

Return the set of `n`-th roots of this group element.

EXAMPLES:

```
sage: from sage.groups.libgap_group import GroupLibGAP
sage: G = GroupLibGAP(libgap.GL(3,3))
sage: a,b = G.gens()
sage: g = a*b**2*a*~b
sage: r = g.nth_roots(4)
sage: r
[[ [ Z(3), Z(3), Z(3)^0 ], [ Z(3)^0, Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3), 0*Z(3)␣
↪] ],
 [ [ Z(3)^0, Z(3)^0, Z(3) ], [ Z(3), Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, 0*Z(3)␣
↪] ]]
sage: r[0]**4 == r[1]**4 == g
True
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_group import GroupLibGAP
>>> G = GroupLibGAP(libgap.GL(Integer(3),Integer(3)))
>>> a,b = G.gens()
>>> g = a*b**Integer(2)*a*~b
>>> r = g.nth_roots(Integer(4))
>>> r
[[ [ Z(3), Z(3), Z(3)^0 ], [ Z(3)^0, Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3), 0*Z(3)␣
↪] ],
 [ [ Z(3)^0, Z(3)^0, Z(3) ], [ Z(3), Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, 0*Z(3)␣
↪] ]]
>>> r[Integer(0)]**Integer(4) == r[Integer(1)]**Integer(4) == g
True
```

order()

Return the multiplicative order.

EXAMPLES:

```
sage: from sage.groups.libgap_group import GroupLibGAP
sage: G = GroupLibGAP(libgap.GL(2, 3))
sage: a,b = G.gens()
sage: print(a.order())
2
sage: print(a.multiplicative_order())
2

sage: z = Mod(0, 3)
sage: o = Mod(1, 3)
sage: G(libgap([[o,o],[z,o]])).order()
3
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_group import GroupLibGAP
>>> G = GroupLibGAP(libgap.GL(Integer(2), Integer(3)))
>>> a,b = G.gens()
>>> print(a.order())
2
>>> print(a.multiplicative_order())
2
```

(continues on next page)

(continued from previous page)

```
>>> z = Mod(Integer(0), Integer(3))
>>> o = Mod(Integer(1), Integer(3))
>>> G(libgap([[o,o],[z,o]])).order()
3
```

class sage.groups.libgap_wrapper.**ParentLibGAP**(*libgap_parent*, *ambient=None*)

Bases: SageObject

A class for parents to keep track of the GAP parent.

This is not a complete group in Sage, this class is only a base class that you can use to implement your own groups with LibGAP. See *libgap_group* for a minimal example of a group that is actually usable.

Your implementation definitely needs to supply

- `__reduce__()`: serialize the LibGAP group. Since GAP does not support Python pickles natively, you need to figure out yourself how you can recreate the group from a pickle.

INPUT:

- `libgap_parent` – the libgap element that is the parent in GAP
- `ambient` – a derived class of *ParentLibGAP* or `None` (default); the ambient class if `libgap_parent` has been defined as a subgroup

EXAMPLES:

```
sage: from sage.groups.libgap_wrapper import ElementLibGAP, ParentLibGAP
sage: from sage.groups.group import Group
sage: class FooElement(ElementLibGAP):
....:     pass
sage: class FooGroup(Group, ParentLibGAP):
....:     Element = FooElement
....:     def __init__(self):
....:         lg = libgap(libgap.CyclicGroup(3))    # dummy
....:         ParentLibGAP.__init__(self, lg)
....:         Group.__init__(self)
sage: FooGroup()
<pc group of size 3 with 1 generator>
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_wrapper import ElementLibGAP, ParentLibGAP
>>> from sage.groups.group import Group
>>> class FooElement(ElementLibGAP):
...     pass
>>> class FooGroup(Group, ParentLibGAP):
...     Element = FooElement
...     def __init__(self):
...         lg = libgap(libgap.CyclicGroup(Integer(3)))    # dummy
...         ParentLibGAP.__init__(self, lg)
...         Group.__init__(self)
>>> FooGroup()
<pc group of size 3 with 1 generator>
```

ambient()

Return the ambient group of a subgroup.

OUTPUT:

A group containing `self`. If `self` has not been defined as a subgroup, we just return `self`.

EXAMPLES:

```
sage: G = FreeGroup(3)
sage: G.ambient() is G
True
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(3))
>>> G.ambient() is G
True
```

gap()

Return the GAP representation of `self`.

OUTPUT: a `GapElement`

EXAMPLES:

```
sage: G = FreeGroup(3); G
Free Group on generators {x0, x1, x2}
sage: G.gap()
<free group on the generators [ x0, x1, x2 ]>
sage: G.gap().parent()
C library interface to GAP
sage: type(G.gap())
<class 'sage.libs.gap.element.GapElement'>
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(3)); G
Free Group on generators {x0, x1, x2}
>>> G.gap()
<free group on the generators [ x0, x1, x2 ]>
>>> G.gap().parent()
C library interface to GAP
>>> type(G.gap())
<class 'sage.libs.gap.element.GapElement'>
```

This can be useful, for example, to call GAP functions that are not wrapped in Sage:

```
sage: G = FreeGroup(3)
sage: H = G.gap()
sage: H.DirectProduct(H)
<fp group on the generators [ f1, f2, f3, f4, f5, f6 ]>
sage: H.DirectProduct(H).RelatorsOfFpGroup()
[ f1^-1*f4^-1*f1*f4, f1^-1*f5^-1*f1*f5, f1^-1*f6^-1*f1*f6, f2^-1*f4^-1*f2*f4,
  f2^-1*f5^-1*f2*f5, f2^-1*f6^-1*f2*f6, f3^-1*f4^-1*f3*f4, f3^-1*f5^-1*f3*f5,
  f3^-1*f6^-1*f3*f6 ]
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(3))
>>> H = G.gap()
>>> H.DirectProduct(H)
<fp group on the generators [ f1, f2, f3, f4, f5, f6 ]>
>>> H.DirectProduct(H).RelatorsOfFpGroup()
[ f1^-1*f4^-1*f1*f4, f1^-1*f5^-1*f1*f5, f1^-1*f6^-1*f1*f6, f2^-1*f4^-1*f2*f4,
  f2^-1*f5^-1*f2*f5, f2^-1*f6^-1*f2*f6, f3^-1*f4^-1*f3*f4, f3^-1*f5^-1*f3*f5,
  f3^-1*f6^-1*f3*f6 ]
```

We can also convert directly to libgap:

```
sage: libgap(GL(2, ZZ))
GL(2,Integers)
```

```
>>> from sage.all import *
>>> libgap(GL(Integer(2), ZZ))
GL(2,Integers)
```

gen (i)

Return the i-th generator of `self`.

> **Warning**
>
> Indexing starts at 0 as usual in Sage/Python. Not as in GAP, where indexing starts at 1.

INPUT:

- `i` – integer between 0 (inclusive) and *ngens()* (exclusive); the index of the generator

OUTPUT: the i-th generator of the group

EXAMPLES:

```
sage: G = FreeGroup('a, b')
sage: G.gen(0)
a
sage: G.gen(1)
b
```

```
>>> from sage.all import *
>>> G = FreeGroup('a, b')
>>> G.gen(Integer(0))
a
>>> G.gen(Integer(1))
b
```

generators ()

Return the generators of the group.

EXAMPLES:

```
sage: G = FreeGroup(2)
sage: G.gens()
(x0, x1)
sage: H = FreeGroup('a, b, c')
sage: H.gens()
(a, b, c)
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(2))
>>> G.gens()
(x0, x1)
>>> H = FreeGroup('a, b, c')
>>> H.gens()
(a, b, c)
```

`generators()` is an alias for `gens()`

```
sage: G = FreeGroup('a, b')
sage: G.generators()
(a, b)
sage: H = FreeGroup(3, 'x')
sage: H.generators()
(x0, x1, x2)
```

```
>>> from sage.all import *
>>> G = FreeGroup('a, b')
>>> G.generators()
(a, b)
>>> H = FreeGroup(Integer(3), 'x')
>>> H.generators()
(x0, x1, x2)
```

gens()

Return the generators of the group.

EXAMPLES:

```
sage: G = FreeGroup(2)
sage: G.gens()
(x0, x1)
sage: H = FreeGroup('a, b, c')
sage: H.gens()
(a, b, c)
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(2))
>>> G.gens()
(x0, x1)
>>> H = FreeGroup('a, b, c')
>>> H.gens()
(a, b, c)
```

`generators()` is an alias for `gens()`

```
sage: G = FreeGroup('a, b')
sage: G.generators()
(a, b)
sage: H = FreeGroup(3, 'x')
sage: H.generators()
(x0, x1, x2)
```

```
>>> from sage.all import *
>>> G = FreeGroup('a, b')
>>> G.generators()
(a, b)
>>> H = FreeGroup(Integer(3), 'x')
>>> H.generators()
(x0, x1, x2)
```

is_subgroup()

Return whether the group was defined as a subgroup of a bigger group.

You can access the containing group with `ambient()`.

OUTPUT: boolean

EXAMPLES:

```
sage: G = FreeGroup(3)
sage: G.is_subgroup()
False
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(3))
>>> G.is_subgroup()
False
```

maximal_normal_subgroups()

Return the maximal proper normal subgroups of `self`.

This raises an error if $G/[G,G]$ is infinite, yielding infinitely many maximal normal subgroups.

EXAMPLES:

```
sage: SL(2,GF(49)).minimal_normal_subgroups()
[Subgroup with 1 generators (
 [6 0]
 [0 6]
 ) of Special Linear Group of degree 2 over Finite Field in z2 of size 7^2]
```

```
>>> from sage.all import *
>>> SL(Integer(2),GF(Integer(49))).minimal_normal_subgroups()
[Subgroup with 1 generators (
 [6 0]
 [0 6]
 ) of Special Linear Group of degree 2 over Finite Field in z2 of size 7^2]
```

minimal_normal_subgroups()

Return the nontrivial minimal normal subgroups of `self`.

EXAMPLES:

```
sage: SL(2,GF(49)).minimal_normal_subgroups()
[Subgroup with 1 generators (
 [6 0]
 [0 6]
 ) of Special Linear Group of degree 2 over Finite Field in z2 of size 7^2]
```

```
>>> from sage.all import *
>>> SL(Integer(2),GF(Integer(49))).minimal_normal_subgroups()
[Subgroup with 1 generators (
 [6 0]
 [0 6]
 ) of Special Linear Group of degree 2 over Finite Field in z2 of size 7^2]
```

ngens()

Return the number of generators of `self`.

OUTPUT: integer

EXAMPLES:

```
sage: G = FreeGroup(2)
sage: G.ngens()
2
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(2))
>>> G.ngens()
2
```

one()

Return the identity element of `self`.

EXAMPLES:

```
sage: G = FreeGroup(3)
sage: G.one()
1
sage: G.one() == G([])
True
sage: G.one().Tietze()
()
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(3))
>>> G.one()
1
>>> G.one() == G([])
True
>>> G.one().Tietze()
()
```

subgroup(*generators*)

Return the subgroup generated.

INPUT:

- `generators` – list/tuple/iterable of group elements

OUTPUT: the subgroup generated by `generators`

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G = F.subgroup([a^2*b]);   G
Group([ a^2*b ])
sage: G.gens()
(a^2*b,)
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('a', 'b',)); (a, b,) = F._first_ngens(2)
>>> G = F.subgroup([a**Integer(2)*b]);   G
Group([ a^2*b ])
>>> G.gens()
(a^2*b,)
```

We check that coercions between the subgroup and its ambient group work:

```
sage: F.0 * G.0
a^3*b
```

```
>>> from sage.all import *
>>> F.gen(0) * G.gen(0)
a^3*b
```

Checking that Issue #19270 is fixed:

```
sage: gens = [w.matrix() for w in WeylGroup(['B', 3])]
sage: G = MatrixGroup(gens)
sage: import itertools
sage: diagonals = itertools.product((1,-1), repeat=3)
sage: subgroup_gens = [diagonal_matrix(L) for L in diagonals]
sage: G.subgroup(subgroup_gens)
Subgroup with 8 generators of Matrix group over Rational Field with 48␣
↪generators
```

```
>>> from sage.all import *
>>> gens = [w.matrix() for w in WeylGroup(['B', Integer(3)])]
>>> G = MatrixGroup(gens)
>>> import itertools
>>> diagonals = itertools.product((Integer(1),-Integer(1)), repeat=Integer(3))
>>> subgroup_gens = [diagonal_matrix(L) for L in diagonals]
>>> G.subgroup(subgroup_gens)
Subgroup with 8 generators of Matrix group over Rational Field with 48␣
↪generators
```

CHAPTER

FIVE

GENERIC LIBGAP-BASED GROUP

This is useful if you need to use a GAP group implementation in Sage that does not have a dedicated Sage interface.

If you want to implement your own group class, you should not derive from this but directly from `ParentLibGAP`.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G_gap = libgap.Group([ (a*b^2).gap() ])
sage: from sage.groups.libgap_group import GroupLibGAP
sage: G = GroupLibGAP(G_gap);  G
Group([ a*b^2 ])
sage: type(G)
<class 'sage.groups.libgap_group.GroupLibGAP_with_category'>
sage: G.gens()
(a*b^2,)
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('a', 'b',)); (a, b,) = F._first_ngens(2)
>>> G_gap = libgap.Group([ (a*b**Integer(2)).gap() ])
>>> from sage.groups.libgap_group import GroupLibGAP
>>> G = GroupLibGAP(G_gap);  G
Group([ a*b^2 ])
>>> type(G)
<class 'sage.groups.libgap_group.GroupLibGAP_with_category'>
>>> G.gens()
(a*b^2,)
```

class sage.groups.libgap_group.**GroupLibGAP**(**args*, ***kwds*)

Bases: `GroupMixinLibGAP`, `Group`, `ParentLibGAP`

Group interface for LibGAP-based groups.

INPUT:

Same as `ParentLibGAP`.

Element

alias of `ElementLibGAP`

CHAPTER

SIX

MIX-IN CLASS FOR GAP-BASED GROUPS

This class adds access to GAP functionality to groups such that parent and element have a `gap()` method that returns a GAP object for the parent/element.

If your group implementation uses libgap, then you should add *GroupMixinLibGAP* as the first class that you are deriving from. This ensures that it properly overrides any default methods that just raise `NotImplementedError`.

class `sage.groups.libgap_mixin.`**`GroupMixinLibGAP`**

Bases: `object`

`cardinality``()`

Implement `EnumeratedSets.ParentMethods.cardinality()`.

EXAMPLES:

```
sage: G = Sp(4,GF(3))
sage: G.cardinality()
51840

sage: G = SL(4,GF(3))
sage: G.cardinality()
12130560

sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,2],[-1,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.cardinality()
480

sage: G = MatrixGroup([matrix(ZZ,2,[1,1,0,1])])
sage: G.cardinality()
+Infinity

sage: G = Sp(4,GF(3))
sage: G.cardinality()
51840

sage: G = SL(4,GF(3))
sage: G.cardinality()
12130560

sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,2],[-1,1]]),MS([[1,1],[0,1]])]
```

(continues on next page)

(continued from previous page)

```
sage: G = MatrixGroup(gens)
sage: G.cardinality()
480

sage: G = MatrixGroup([matrix(ZZ,2,[1,1,0,1])])
sage: G.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> G = Sp(Integer(4),GF(Integer(3)))
>>> G.cardinality()
51840

>>> G = SL(Integer(4),GF(Integer(3)))
>>> G.cardinality()
12130560

>>> F = GF(Integer(5)); MS = MatrixSpace(F,Integer(2),Integer(2))
>>> gens = [MS([[Integer(1),Integer(2)],[-Integer(1),Integer(1)]]),
↪MS([[Integer(1),Integer(1)],[Integer(0),Integer(1)]])]
>>> G = MatrixGroup(gens)
>>> G.cardinality()
480

>>> G = MatrixGroup([matrix(ZZ,Integer(2),[Integer(1),Integer(1),Integer(0),
↪Integer(1)])])
>>> G.cardinality()
+Infinity

>>> G = Sp(Integer(4),GF(Integer(3)))
>>> G.cardinality()
51840

>>> G = SL(Integer(4),GF(Integer(3)))
>>> G.cardinality()
12130560

>>> F = GF(Integer(5)); MS = MatrixSpace(F,Integer(2),Integer(2))
>>> gens = [MS([[Integer(1),Integer(2)],[-Integer(1),Integer(1)]]),
↪MS([[Integer(1),Integer(1)],[Integer(0),Integer(1)]])]
>>> G = MatrixGroup(gens)
>>> G.cardinality()
480

>>> G = MatrixGroup([matrix(ZZ,Integer(2),[Integer(1),Integer(1),Integer(0),
↪Integer(1)])])
>>> G.cardinality()
+Infinity
```

center()

Return the center of this group as a subgroup.

OUTPUT: the center as a subgroup

EXAMPLES:

```
sage: G = SU(3, GF(2))                                                          #␣
↪needs sage.rings.finite_rings
sage: G.center()                                                                #␣
↪needs sage.rings.finite_rings
Subgroup with 1 generators (
[a 0 0]
[0 a 0]
[0 0 a]
) of Special Unitary Group of degree 3 over Finite Field in a of size 2^2
sage: GL(2, GF(3)).center()
Subgroup with 1 generators (
[2 0]
[0 2]
) of General Linear Group of degree 2 over Finite Field of size 3
sage: GL(3, GF(3)).center()
Subgroup with 1 generators (
[2 0 0]
[0 2 0]
[0 0 2]
) of General Linear Group of degree 3 over Finite Field of size 3
sage: GU(3, GF(2)).center()                                                     #␣
↪needs sage.rings.finite_rings
Subgroup with 1 generators (
[a + 1     0     0]
[    0 a + 1     0]
[    0     0 a + 1]
) of General Unitary Group of degree 3 over Finite Field in a of size 2^2

sage: A = Matrix(FiniteField(5), [[2,0,0], [0,3,0], [0,0,1]])
sage: B = Matrix(FiniteField(5), [[1,0,0], [0,1,0], [0,1,1]])
sage: MatrixGroup([A,B]).center()
Subgroup with 1 generators (
[1 0 0]
[0 1 0]
[0 0 1]
) of Matrix group over Finite Field of size 5 with 2 generators (
[2 0 0]  [1 0 0]
[0 3 0]  [0 1 0]
[0 0 1], [0 1 1]
)

sage: GL = groups.matrix.GL(3, ZZ)
sage: GL.center()
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

```
>>> from sage.all import *
>>> G = SU(Integer(3), GF(Integer(2)))                                          ␣
↪               # needs sage.rings.finite_rings
>>> G.center()                                                                #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.finite_rings
Subgroup with 1 generators (
[a 0 0]
[0 a 0]
[0 0 a]
) of Special Unitary Group of degree 3 over Finite Field in a of size 2^2
>>> GL(Integer(2), GF(Integer(3))).center()
Subgroup with 1 generators (
[2 0]
[0 2]
) of General Linear Group of degree 2 over Finite Field of size 3
>>> GL(Integer(3), GF(Integer(3))).center()
Subgroup with 1 generators (
[2 0 0]
[0 2 0]
[0 0 2]
) of General Linear Group of degree 3 over Finite Field of size 3
>>> GU(Integer(3), GF(Integer(2))).center()                                    ␣
↪                # needs sage.rings.finite_rings
Subgroup with 1 generators (
[a + 1     0     0]
[    0 a + 1     0]
[    0     0 a + 1]
) of General Unitary Group of degree 3 over Finite Field in a of size 2^2

>>> A = Matrix(FiniteField(Integer(5)), [[Integer(2),Integer(0),Integer(0)],␣
↪[Integer(0),Integer(3),Integer(0)], [Integer(0),Integer(0),Integer(1)]])
>>> B = Matrix(FiniteField(Integer(5)), [[Integer(1),Integer(0),Integer(0)],␣
↪[Integer(0),Integer(1),Integer(0)], [Integer(0),Integer(1),Integer(1)]])
>>> MatrixGroup([A,B]).center()
Subgroup with 1 generators (
[1 0 0]
[0 1 0]
[0 0 1]
) of Matrix group over Finite Field of size 5 with 2 generators (
[2 0 0]  [1 0 0]
[0 3 0]  [0 1 0]
[0 0 1], [0 1 1]
)

>>> GL = groups.matrix.GL(Integer(3), ZZ)
>>> GL.center()
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

centralizer(*g*)

Return the centralizer of `g` in `self`.

EXAMPLES:

```
sage: G = groups.matrix.GL(2, 3)
```

(continues on next page)

(continued from previous page)

```
sage: g = G([[1,1], [1,0]])
sage: C = G.centralizer(g); C
Subgroup with 3 generators (
[1 1]  [2 0]  [2 1]
[1 0], [0 2], [1 1]
) of General Linear Group of degree 2 over Finite Field of size 3
sage: C.order()
8

sage: S = G.subgroup([G([[2,0],[0,2]]), G([[0,1],[2,0]])]); S
Subgroup with 2 generators (
[2 0]  [0 1]
[0 2], [2 0]
) of General Linear Group of degree 2 over Finite Field of size 3
sage: G.centralizer(S)
Subgroup with 3 generators (
[2 0]  [0 1]  [2 2]
[0 2], [2 0], [1 2]
) of General Linear Group of degree 2 over Finite Field of size 3
sage: G = GL(3,2)
sage: all(G.order() == G.centralizer(x).order() * G.conjugacy_class(x).
↪cardinality()
....:     for x in G)
True
sage: H = groups.matrix.Heisenberg(2)
sage: H.centralizer(H.an_element())
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

```
>>> from sage.all import *
>>> G = groups.matrix.GL(Integer(2), Integer(3))
>>> g = G([[Integer(1),Integer(1)], [Integer(1),Integer(0)]])
>>> C = G.centralizer(g); C
Subgroup with 3 generators (
[1 1]  [2 0]  [2 1]
[1 0], [0 2], [1 1]
) of General Linear Group of degree 2 over Finite Field of size 3
>>> C.order()
8

>>> S = G.subgroup([G([[Integer(2),Integer(0)],[Integer(0),Integer(2)]]),␣
↪G([[Integer(0),Integer(1)],[Integer(2),Integer(0)]])]); S
Subgroup with 2 generators (
[2 0]  [0 1]
[0 2], [2 0]
) of General Linear Group of degree 2 over Finite Field of size 3
>>> G.centralizer(S)
Subgroup with 3 generators (
[2 0]  [0 1]  [2 2]
[0 2], [2 0], [1 2]
) of General Linear Group of degree 2 over Finite Field of size 3
```

(continues on next page)

(continued from previous page)

```
>>> G = GL(Integer(3),Integer(2))
>>> all(G.order() == G.centralizer(x).order() * G.conjugacy_class(x).
↪cardinality()
...     for x in G)
True
>>> H = groups.matrix.Heisenberg(Integer(2))
>>> H.centralizer(H.an_element())
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

character(*values*)

Return a group character from `values`, where `values` is a list of the values of the character evaluated on the conjugacy classes.

INPUT:

- `values` – list of values of the character

OUTPUT: a group character

EXAMPLES:

```
sage: G = MatrixGroup(AlternatingGroup(4))
sage: G.character([1]*len(G.conjugacy_classes_representatives()))           #␣
↪needs sage.rings.number_field
Character of Matrix group over Integer Ring with 12 generators
```

```
>>> from sage.all import *
>>> G = MatrixGroup(AlternatingGroup(Integer(4)))
>>> G.character([Integer(1)]*len(G.conjugacy_classes_representatives()))    ␣
↪     # needs sage.rings.number_field
Character of Matrix group over Integer Ring with 12 generators
```

```
sage: G = GL(2,ZZ)
sage: G.character([1,1,1,1])
Traceback (most recent call last):
...
NotImplementedError: only implemented for finite groups
```

```
>>> from sage.all import *
>>> G = GL(Integer(2),ZZ)
>>> G.character([Integer(1),Integer(1),Integer(1),Integer(1)])
Traceback (most recent call last):
...
NotImplementedError: only implemented for finite groups
```

character_table()

Return the matrix of values of the irreducible characters of this group G at its conjugacy classes.

The columns represent the conjugacy classes of G and the rows represent the different irreducible characters in the ordering given by GAP.

OUTPUT: a matrix defined over a cyclotomic field

EXAMPLES:

```
sage: MatrixGroup(SymmetricGroup(2)).character_table()                    #␣
↪needs sage.rings.number_field
[ 1 -1]
[ 1  1]
sage: MatrixGroup(SymmetricGroup(3)).character_table()                    #␣
↪needs sage.rings.number_field
[ 1  1 -1]
[ 2 -1  0]
[ 1  1  1]
sage: MatrixGroup(SymmetricGroup(5)).character_table()   # long time
[ 1 -1 -1  1 -1  1  1]
[ 4  0  1 -1 -2  1  0]
[ 5  1 -1  0 -1 -1  1]
[ 6  0  0  1  0  0 -2]
[ 5 -1  1  0  1 -1  1]
[ 4  0 -1 -1  2  1  0]
[ 1  1  1  1  1  1  1]
```

```
>>> from sage.all import *
>>> MatrixGroup(SymmetricGroup(Integer(2))).character_table()              ␣
↪      # needs sage.rings.number_field
[ 1 -1]
[ 1  1]
>>> MatrixGroup(SymmetricGroup(Integer(3))).character_table()              ␣
↪      # needs sage.rings.number_field
[ 1  1 -1]
[ 2 -1  0]
[ 1  1  1]
>>> MatrixGroup(SymmetricGroup(Integer(5))).character_table()   # long time
[ 1 -1 -1  1 -1  1  1]
[ 4  0  1 -1 -2  1  0]
[ 5  1 -1  0 -1 -1  1]
[ 6  0  0  1  0  0 -2]
[ 5 -1  1  0  1 -1  1]
[ 4  0 -1 -1  2  1  0]
[ 1  1  1  1  1  1  1]
```

class_function(*values*)

Return the class function with given values.

INPUT:

- `values` – list/tuple/iterable of numbers; the values of the class function on the conjugacy classes, in that order

EXAMPLES:

```
sage: G = GL(2,GF(3))
sage: chi = G.class_function(range(8))                                    #␣
↪needs sage.rings.number_field
sage: list(chi)                                                           #␣
↪needs sage.rings.number_field
[0, 1, 2, 3, 4, 5, 6, 7]
```

```
>>> from sage.all import *
>>> G = GL(Integer(2),GF(Integer(3)))
>>> chi = G.class_function(range(Integer(8)))                                      ␣
↪     # needs sage.rings.number_field
>>> list(chi)                                                                     #␣
↪needs sage.rings.number_field
[0, 1, 2, 3, 4, 5, 6, 7]
```

conjugacy_class (g)

Return the conjugacy class of g.

OUTPUT:

The conjugacy class of g in the group self. If self is the group denoted by G, this method computes the set $\{x^{-1}gx \mid x \in G\}$.

EXAMPLES:

```
sage: G = SL(2, QQ)
sage: g = G([[1,1],[0,1]])
sage: G.conjugacy_class(g)
Conjugacy class of [1 1]
[0 1] in Special Linear Group of degree 2 over Rational Field
```

```
>>> from sage.all import *
>>> G = SL(Integer(2), QQ)
>>> g = G([[Integer(1),Integer(1)],[Integer(0),Integer(1)]])
>>> G.conjugacy_class(g)
Conjugacy class of [1 1]
[0 1] in Special Linear Group of degree 2 over Rational Field
```

conjugacy_classes ()

Return a list with all the conjugacy classes of self.

EXAMPLES:

```
sage: G = SL(2, GF(2))
sage: G.conjugacy_classes()
(Conjugacy class of [1 0]
 [0 1] in Special Linear Group of degree 2 over Finite Field of size 2,
 Conjugacy class of [0 1]
 [1 0] in Special Linear Group of degree 2 over Finite Field of size 2,
 Conjugacy class of [0 1]
 [1 1] in Special Linear Group of degree 2 over Finite Field of size 2)
```

```
>>> from sage.all import *
>>> G = SL(Integer(2), GF(Integer(2)))
>>> G.conjugacy_classes()
(Conjugacy class of [1 0]
 [0 1] in Special Linear Group of degree 2 over Finite Field of size 2,
 Conjugacy class of [0 1]
 [1 0] in Special Linear Group of degree 2 over Finite Field of size 2,
 Conjugacy class of [0 1]
 [1 1] in Special Linear Group of degree 2 over Finite Field of size 2)
```

```
sage: GL(2,ZZ).conjugacy_classes()
Traceback (most recent call last):
...
NotImplementedError: only implemented for finite groups
```

```
>>> from sage.all import *
>>> GL(Integer(2),ZZ).conjugacy_classes()
Traceback (most recent call last):
...
NotImplementedError: only implemented for finite groups
```

conjugacy_classes_representatives()

Return a set of representatives for each of the conjugacy classes of the group.

EXAMPLES:

```
sage: G = SU(3,GF(2))                                                          #␣
↪needs sage.rings.finite_rings
sage: len(G.conjugacy_classes_representatives())                               #␣
↪needs sage.rings.finite_rings
16

sage: G = GL(2,GF(3))
sage: G.conjugacy_classes_representatives()
(
[1 0]  [0 2]  [2 0]  [0 2]  [0 2]  [0 1]  [0 1]  [2 0]
[0 1], [1 1], [0 2], [1 2], [1 0], [1 2], [1 1], [0 1]
)

sage: len(GU(2,GF(5)).conjugacy_classes_representatives())                     #␣
↪needs sage.rings.finite_rings
36
```

```
>>> from sage.all import *
>>> G = SU(Integer(3),GF(Integer(2)))                                            ␣
↪               # needs sage.rings.finite_rings
>>> len(G.conjugacy_classes_representatives())                                 #␣
↪needs sage.rings.finite_rings
16

>>> G = GL(Integer(2),GF(Integer(3)))
>>> G.conjugacy_classes_representatives()
(
[1 0]  [0 2]  [2 0]  [0 2]  [0 2]  [0 1]  [0 1]  [2 0]
[0 1], [1 1], [0 2], [1 2], [1 0], [1 2], [1 1], [0 1]
)

>>> len(GU(Integer(2),GF(Integer(5))).conjugacy_classes_representatives())    ␣
↪               # needs sage.rings.finite_rings
36
```

```
sage: GL(2,ZZ).conjugacy_classes_representatives()
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
NotImplementedError: only implemented for finite groups
```

```
>>> from sage.all import *
>>> GL(Integer(2),ZZ).conjugacy_classes_representatives()
Traceback (most recent call last):
...
NotImplementedError: only implemented for finite groups
```

conjugacy_classes_subgroups()

Return a complete list of representatives of conjugacy classes of subgroups in `self`.

The ordering is that given by GAP.

EXAMPLES:

```
sage: G = groups.matrix.GL(2,2)
sage: G.conjugacy_classes_subgroups()
[Subgroup with 0 generators ()
   of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 1 generators (
 [1 1]
 [0 1]
 ) of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 1 generators (
 [0 1]
 [1 1]
 ) of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 2 generators (
 [0 1]  [1 1]
 [1 1], [0 1]
 ) of General Linear Group of degree 2 over Finite Field of size 2]

sage: H = groups.matrix.Heisenberg(2)
sage: H.conjugacy_classes_subgroups()
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

```
>>> from sage.all import *
>>> G = groups.matrix.GL(Integer(2),Integer(2))
>>> G.conjugacy_classes_subgroups()
[Subgroup with 0 generators ()
   of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 1 generators (
 [1 1]
 [0 1]
 ) of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 1 generators (
 [0 1]
 [1 1]
```

(continues on next page)

(continued from previous page)

```
 ) of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 2 generators (
 [0 1]  [1 1]
 [1 1], [0 1]
 ) of General Linear Group of degree 2 over Finite Field of size 2]

>>> H = groups.matrix.Heisenberg(Integer(2))
>>> H.conjugacy_classes_subgroups()
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

exponent()

Compute the exponent of the group.

The exponent e of a group G is the LCM of the orders of its elements, that is, e is the smallest integer such that $g^e = 1$ for all $g \in G$.

EXAMPLES:

```
sage: G = groups.matrix.GL(2, 3)
sage: G.exponent()
24

sage: H = groups.matrix.Heisenberg(2)
sage: H.exponent()
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

```
>>> from sage.all import *
>>> G = groups.matrix.GL(Integer(2), Integer(3))
>>> G.exponent()
24

>>> H = groups.matrix.Heisenberg(Integer(2))
>>> H.exponent()
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

group_id()

Return the ID code of `self`, which is a list of two integers.

It is a unique identified assigned by GAP for groups in the `SmallGroup` library.

EXAMPLES:

```
sage: PGL(2,3).group_id()
[24, 12]
sage: SymmetricGroup(4).group_id()
[24, 12]

sage: G = groups.matrix.GL(2, 2)
```

(continues on next page)

(continued from previous page)

```
sage: G.group_id()
[6, 1]
sage: G = groups.matrix.GL(2, 3)
sage: G.id()
[48, 29]

sage: G = groups.matrix.GL(2, ZZ)
sage: G.group_id()
Traceback (most recent call last):
...
GAPError: Error, the group identification for groups of size infinity is not␣
↪available
```

```
>>> from sage.all import *
>>> PGL(Integer(2),Integer(3)).group_id()
[24, 12]
>>> SymmetricGroup(Integer(4)).group_id()
[24, 12]

>>> G = groups.matrix.GL(Integer(2), Integer(2))
>>> G.group_id()
[6, 1]
>>> G = groups.matrix.GL(Integer(2), Integer(3))
>>> G.id()
[48, 29]

>>> G = groups.matrix.GL(Integer(2), ZZ)
>>> G.group_id()
Traceback (most recent call last):
...
GAPError: Error, the group identification for groups of size infinity is not␣
↪available
```

id()

Return the ID code of `self`, which is a list of two integers.

It is a unique identified assigned by GAP for groups in the `SmallGroup` library.

EXAMPLES:

```
sage: PGL(2,3).group_id()
[24, 12]
sage: SymmetricGroup(4).group_id()
[24, 12]

sage: G = groups.matrix.GL(2, 2)
sage: G.group_id()
[6, 1]
sage: G = groups.matrix.GL(2, 3)
sage: G.id()
[48, 29]

sage: G = groups.matrix.GL(2, ZZ)
```

(continues on next page)

(continued from previous page)

```
sage: G.group_id()
Traceback (most recent call last):
...
GAPError: Error, the group identification for groups of size infinity is not␣
↪available
```

```
>>> from sage.all import *
>>> PGL(Integer(2),Integer(3)).group_id()
[24, 12]
>>> SymmetricGroup(Integer(4)).group_id()
[24, 12]

>>> G = groups.matrix.GL(Integer(2), Integer(2))
>>> G.group_id()
[6, 1]
>>> G = groups.matrix.GL(Integer(2), Integer(3))
>>> G.id()
[48, 29]

>>> G = groups.matrix.GL(Integer(2), ZZ)
>>> G.group_id()
Traceback (most recent call last):
...
GAPError: Error, the group identification for groups of size infinity is not␣
↪available
```

intersection(*other*)

Return the intersection of two groups (if it makes sense) as a subgroup of the first group.

EXAMPLES:

```
sage: A = Matrix([(0, 1/2, 0), (2, 0, 0), (0, 0, 1)])
sage: B = Matrix([(0, 1/2, 0), (-2, -1, 2), (0, 0, 1)])
sage: G = MatrixGroup([A,B])
sage: len(G)   # isomorphic to S_3
6
sage: G.intersection(GL(3,ZZ))
Subgroup with 1 generators (
[ 1  0  0]
[-2 -1  2]
[ 0  0  1]
) of Matrix group over Rational Field with 2 generators (
  [  0 1/2   0]  [  0 1/2   0]
  [  2   0   0]  [ -2  -1   2]
  [  0   0   1], [  0   0   1]
  )
sage: GL(3,ZZ).intersection(G)
Subgroup with 1 generators (
[ 1  0  0]
[-2 -1  2]
[ 0  0  1]
) of General Linear Group of degree 3 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
sage: G.intersection(SL(3,ZZ))
Subgroup with 0 generators ()
  of Matrix group over Rational Field with 2 generators (
  [  0 1/2   0]  [  0 1/2   0]
  [  2   0   0]  [ -2  -1   2]
  [  0   0   1], [  0   0   1]
  )
```

```
>>> from sage.all import *
>>> A = Matrix([(Integer(0), Integer(1)/Integer(2), Integer(0)), (Integer(2),␣
↪Integer(0), Integer(0)), (Integer(0), Integer(0), Integer(1))])
>>> B = Matrix([(Integer(0), Integer(1)/Integer(2), Integer(0)), (-Integer(2),
↪ -Integer(1), Integer(2)), (Integer(0), Integer(0), Integer(1))])
>>> G = MatrixGroup([A,B])
>>> len(G)   # isomorphic to S_3
6
>>> G.intersection(GL(Integer(3),ZZ))
Subgroup with 1 generators (
[ 1  0  0]
[-2 -1  2]
[ 0  0  1]
) of Matrix group over Rational Field with 2 generators (
  [  0 1/2   0]  [  0 1/2   0]
  [  2   0   0]  [ -2  -1   2]
  [  0   0   1], [  0   0   1]
  )
>>> GL(Integer(3),ZZ).intersection(G)
Subgroup with 1 generators (
[ 1  0  0]
[-2 -1  2]
[ 0  0  1]
) of General Linear Group of degree 3 over Integer Ring
>>> G.intersection(SL(Integer(3),ZZ))
Subgroup with 0 generators ()
  of Matrix group over Rational Field with 2 generators (
  [  0 1/2   0]  [  0 1/2   0]
  [  2   0   0]  [ -2  -1   2]
  [  0   0   1], [  0   0   1]
  )
```

irreducible_characters()

Return the irreducible characters of the group.

OUTPUT: tuple containing all irreducible characters

EXAMPLES:

```
sage: G = GL(2,2)
sage: G.irreducible_characters()                                        #␣
↪needs sage.rings.number_field
(Character of General Linear Group of degree 2 over Finite Field of size 2,
 Character of General Linear Group of degree 2 over Finite Field of size 2,
 Character of General Linear Group of degree 2 over Finite Field of size 2)
```

```
>>> from sage.all import *
>>> G = GL(Integer(2),Integer(2))
>>> G.irreducible_characters()                                          #␣
↪needs sage.rings.number_field
(Character of General Linear Group of degree 2 over Finite Field of size 2,
 Character of General Linear Group of degree 2 over Finite Field of size 2,
 Character of General Linear Group of degree 2 over Finite Field of size 2)
```

```
sage: GL(2,ZZ).irreducible_characters()
Traceback (most recent call last):
...
NotImplementedError: only implemented for finite groups
```

```
>>> from sage.all import *
>>> GL(Integer(2),ZZ).irreducible_characters()
Traceback (most recent call last):
...
NotImplementedError: only implemented for finite groups
```

is_abelian()

Return whether the group is Abelian.

OUTPUT: boolean; `True` if this group is an Abelian group and `False` otherwise

EXAMPLES:

```
sage: from sage.groups.libgap_group import GroupLibGAP
sage: GroupLibGAP(libgap.CyclicGroup(12)).is_abelian()
True
sage: GroupLibGAP(libgap.SymmetricGroup(12)).is_abelian()
False

sage: SL(1, 17).is_abelian()
True
sage: SL(2, 17).is_abelian()
False
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_group import GroupLibGAP
>>> GroupLibGAP(libgap.CyclicGroup(Integer(12))).is_abelian()
True
>>> GroupLibGAP(libgap.SymmetricGroup(Integer(12))).is_abelian()
False

>>> SL(Integer(1), Integer(17)).is_abelian()
True
>>> SL(Integer(2), Integer(17)).is_abelian()
False
```

is_finite()

Test whether the matrix group is finite.

OUTPUT: boolean

EXAMPLES:

```
sage: G = GL(2,GF(3))
sage: G.is_finite()
True
sage: SL(2,ZZ).is_finite()
False
```

```
>>> from sage.all import *
>>> G = GL(Integer(2),GF(Integer(3)))
>>> G.is_finite()
True
>>> SL(Integer(2),ZZ).is_finite()
False
```

is_isomorphic(*H*)

Test whether `self` and `H` are isomorphic groups.

INPUT:

- `H` – a group

OUTPUT: boolean

EXAMPLES:

```
sage: m1 = matrix(GF(3), [[1,1],[0,1]])
sage: m2 = matrix(GF(3), [[1,2],[0,1]])
sage: F = MatrixGroup(m1)
sage: G = MatrixGroup(m1, m2)
sage: H = MatrixGroup(m2)
sage: F.is_isomorphic(G)
True
sage: G.is_isomorphic(H)
True
sage: F.is_isomorphic(H)
True
sage: F == G, G == H, F == H
(False, False, False)
```

```
>>> from sage.all import *
>>> m1 = matrix(GF(Integer(3)), [[Integer(1),Integer(1)],[Integer(0),
↪Integer(1)]])
>>> m2 = matrix(GF(Integer(3)), [[Integer(1),Integer(2)],[Integer(0),
↪Integer(1)]])
>>> F = MatrixGroup(m1)
>>> G = MatrixGroup(m1, m2)
>>> H = MatrixGroup(m2)
>>> F.is_isomorphic(G)
True
>>> G.is_isomorphic(H)
True
>>> F.is_isomorphic(H)
True
>>> F == G, G == H, F == H
(False, False, False)
```

is_nilpotent()

Return whether this group is nilpotent.

EXAMPLES:

```
sage: from sage.groups.libgap_group import GroupLibGAP
sage: GroupLibGAP(libgap.AlternatingGroup(3)).is_nilpotent()
True
sage: GroupLibGAP(libgap.SymmetricGroup(3)).is_nilpotent()
False
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_group import GroupLibGAP
>>> GroupLibGAP(libgap.AlternatingGroup(Integer(3))).is_nilpotent()
True
>>> GroupLibGAP(libgap.SymmetricGroup(Integer(3))).is_nilpotent()
False
```

is_p_group()

Return whether this group is a p-group.

EXAMPLES:

```
sage: from sage.groups.libgap_group import GroupLibGAP
sage: GroupLibGAP(libgap.CyclicGroup(9)).is_p_group()
True
sage: GroupLibGAP(libgap.CyclicGroup(10)).is_p_group()
False
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_group import GroupLibGAP
>>> GroupLibGAP(libgap.CyclicGroup(Integer(9))).is_p_group()
True
>>> GroupLibGAP(libgap.CyclicGroup(Integer(10))).is_p_group()
False
```

is_perfect()

Return whether this group is perfect.

EXAMPLES:

```
sage: from sage.groups.libgap_group import GroupLibGAP
sage: GroupLibGAP(libgap.SymmetricGroup(5)).is_perfect()
False
sage: GroupLibGAP(libgap.AlternatingGroup(5)).is_perfect()
True

sage: SL(3,3).is_perfect()
True
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_group import GroupLibGAP
>>> GroupLibGAP(libgap.SymmetricGroup(Integer(5))).is_perfect()
False
```

(continues on next page)

(continued from previous page)

```
>>> GroupLibGAP(libgap.AlternatingGroup(Integer(5))).is_perfect()
True

>>> SL(Integer(3),Integer(3)).is_perfect()
True
```

is_polycyclic()

Return whether this group is polycyclic.

EXAMPLES:

```
sage: from sage.groups.libgap_group import GroupLibGAP
sage: GroupLibGAP(libgap.AlternatingGroup(4)).is_polycyclic()
True
sage: GroupLibGAP(libgap.AlternatingGroup(5)).is_solvable()
False
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_group import GroupLibGAP
>>> GroupLibGAP(libgap.AlternatingGroup(Integer(4))).is_polycyclic()
True
>>> GroupLibGAP(libgap.AlternatingGroup(Integer(5))).is_solvable()
False
```

is_simple()

Return whether this group is simple.

EXAMPLES:

```
sage: from sage.groups.libgap_group import GroupLibGAP
sage: GroupLibGAP(libgap.SL(2,3)).is_simple()
False
sage: GroupLibGAP(libgap.SL(3,3)).is_simple()
True

sage: SL(3,3).is_simple()
True
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_group import GroupLibGAP
>>> GroupLibGAP(libgap.SL(Integer(2),Integer(3))).is_simple()
False
>>> GroupLibGAP(libgap.SL(Integer(3),Integer(3))).is_simple()
True

>>> SL(Integer(3),Integer(3)).is_simple()
True
```

is_solvable()

Return whether this group is solvable.

EXAMPLES:

```
sage: from sage.groups.libgap_group import GroupLibGAP
sage: GroupLibGAP(libgap.SymmetricGroup(4)).is_solvable()
True
sage: GroupLibGAP(libgap.SymmetricGroup(5)).is_solvable()
False
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_group import GroupLibGAP
>>> GroupLibGAP(libgap.SymmetricGroup(Integer(4))).is_solvable()
True
>>> GroupLibGAP(libgap.SymmetricGroup(Integer(5))).is_solvable()
False
```

is_supersolvable()

Return whether this group is supersolvable.

EXAMPLES:

```
sage: from sage.groups.libgap_group import GroupLibGAP
sage: GroupLibGAP(libgap.SymmetricGroup(3)).is_supersolvable()
True
sage: GroupLibGAP(libgap.SymmetricGroup(4)).is_supersolvable()
False
```

```
>>> from sage.all import *
>>> from sage.groups.libgap_group import GroupLibGAP
>>> GroupLibGAP(libgap.SymmetricGroup(Integer(3))).is_supersolvable()
True
>>> GroupLibGAP(libgap.SymmetricGroup(Integer(4))).is_supersolvable()
False
```

list()

List all elements of this group.

OUTPUT: tuple containing all group elements in a random but fixed order

EXAMPLES:

```
sage: F = GF(3)
sage: gens = [matrix(F,2, [1,0,-1,1]), matrix(F, 2, [1,1,0,1])]
sage: G = MatrixGroup(gens)
sage: G.cardinality()
24
sage: v = G.list()
sage: len(v)
24
sage: v[:5]
(
[1 0]  [2 0]  [0 1]  [0 2]  [1 2]
[0 1], [0 2], [2 0], [1 0], [2 2]
)

sage: all(g in G for g in G.list())
True
```

```
>>> from sage.all import *
>>> F = GF(Integer(3))
>>> gens = [matrix(F,Integer(2), [Integer(1),Integer(0),-Integer(1),
↪Integer(1)]), matrix(F, Integer(2), [Integer(1),Integer(1),Integer(0),
↪Integer(1)])]
>>> G = MatrixGroup(gens)
>>> G.cardinality()
24
>>> v = G.list()
>>> len(v)
24
>>> v[:Integer(5)]
(
[1 0]  [2 0]  [0 1]  [0 2]  [1 2]
[0 1], [0 2], [2 0], [1 0], [2 2]
)

>>> all(g in G for g in G.list())
True
```

An example over a ring (see Issue #5241):

```
sage: M1 = matrix(ZZ,2,[[-1,0],[0,1]])
sage: M2 = matrix(ZZ,2,[[1,0],[0,-1]])
sage: M3 = matrix(ZZ,2,[[-1,0],[0,-1]])
sage: MG = MatrixGroup([M1, M2, M3])
sage: MG.list()
(
[1 0]  [ 1  0]  [-1  0]  [-1  0]
[0 1], [ 0 -1], [ 0  1], [ 0 -1]
)
sage: MG.list()[1]
[ 1  0]
[ 0 -1]
sage: MG.list()[1].parent()
Matrix group over Integer Ring with 3 generators (
[-1  0]  [ 1  0]  [-1  0]
[ 0  1], [ 0 -1], [ 0 -1]
)
```

```
>>> from sage.all import *
>>> M1 = matrix(ZZ,Integer(2),[[-Integer(1),Integer(0)],[Integer(0),
↪Integer(1)]])
>>> M2 = matrix(ZZ,Integer(2),[[Integer(1),Integer(0)],[Integer(0),-
↪Integer(1)]])
>>> M3 = matrix(ZZ,Integer(2),[[-Integer(1),Integer(0)],[Integer(0),-
↪Integer(1)]])
>>> MG = MatrixGroup([M1, M2, M3])
>>> MG.list()
(
[1 0]  [ 1  0]  [-1  0]  [-1  0]
[0 1], [ 0 -1], [ 0  1], [ 0 -1]
)
```

(continues on next page)

(continued from previous page)

```
>>> MG.list()[Integer(1)]
[ 1  0]
[ 0 -1]
>>> MG.list()[Integer(1)].parent()
Matrix group over Integer Ring with 3 generators (
[-1  0]  [ 1  0]  [-1  0]
[ 0  1], [ 0 -1], [ 0 -1]
)
```

An example over a field (see Issue #10515):

```
sage: gens = [matrix(QQ,2,[1,0,0,1])]
sage: MatrixGroup(gens).list()
(
[1 0]
[0 1]
)
```

```
>>> from sage.all import *
>>> gens = [matrix(QQ,Integer(2),[Integer(1),Integer(0),Integer(0),
↪Integer(1)])]
>>> MatrixGroup(gens).list()
(
[1 0]
[0 1]
)
```

Another example over a ring (see Issue #9437):

```
sage: len(SL(2, Zmod(4)).list())
48
```

```
>>> from sage.all import *
>>> len(SL(Integer(2), Zmod(Integer(4))).list())
48
```

An error is raised if the group is not finite:

```
sage: GL(2,ZZ).list()
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

```
>>> from sage.all import *
>>> GL(Integer(2),ZZ).list()
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

order()

Implement `EnumeratedSets.ParentMethods.cardinality()`.

EXAMPLES:

```
sage: G = Sp(4,GF(3))
sage: G.cardinality()
51840

sage: G = SL(4,GF(3))
sage: G.cardinality()
12130560

sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,2],[-1,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.cardinality()
480

sage: G = MatrixGroup([matrix(ZZ,2,[1,1,0,1])])
sage: G.cardinality()
+Infinity

sage: G = Sp(4,GF(3))
sage: G.cardinality()
51840

sage: G = SL(4,GF(3))
sage: G.cardinality()
12130560

sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,2],[-1,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.cardinality()
480

sage: G = MatrixGroup([matrix(ZZ,2,[1,1,0,1])])
sage: G.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> G = Sp(Integer(4),GF(Integer(3)))
>>> G.cardinality()
51840

>>> G = SL(Integer(4),GF(Integer(3)))
>>> G.cardinality()
12130560

>>> F = GF(Integer(5)); MS = MatrixSpace(F,Integer(2),Integer(2))
>>> gens = [MS([[Integer(1),Integer(2)],[-Integer(1),Integer(1)]]),
↪MS([[Integer(1),Integer(1)],[Integer(0),Integer(1)]])]
>>> G = MatrixGroup(gens)
>>> G.cardinality()
480
```

(continues on next page)

(continued from previous page)

```
>>> G = MatrixGroup([matrix(ZZ,Integer(2),[Integer(1),Integer(1),Integer(0),
↪Integer(1)])])
>>> G.cardinality()
+Infinity

>>> G = Sp(Integer(4),GF(Integer(3)))
>>> G.cardinality()
51840

>>> G = SL(Integer(4),GF(Integer(3)))
>>> G.cardinality()
12130560

>>> F = GF(Integer(5)); MS = MatrixSpace(F,Integer(2),Integer(2))
>>> gens = [MS([[Integer(1),Integer(2)],[-Integer(1),Integer(1)]]),
↪MS([[Integer(1),Integer(1)],[Integer(0),Integer(1)]])]
>>> G = MatrixGroup(gens)
>>> G.cardinality()
480

>>> G = MatrixGroup([matrix(ZZ,Integer(2),[Integer(1),Integer(1),Integer(0),
↪Integer(1)])])
>>> G.cardinality()
+Infinity
```

random_element()

Return a random element of this group.

OUTPUT: a group element

EXAMPLES:

```
sage: G = Sp(4,GF(3))
sage: G.random_element()  # random
[2 1 1 1]
[1 0 2 1]
[0 1 1 0]
[1 0 0 1]
sage: G.random_element() in G
True

sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,2],[-1,1]]), MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.random_element()  # random
[1 3]
[0 3]
sage: G.random_element() in G
True
```

```
>>> from sage.all import *
>>> G = Sp(Integer(4),GF(Integer(3)))
>>> G.random_element()  # random
```

(continues on next page)

(continued from previous page)

```
[2 1 1 1]
[1 0 2 1]
[0 1 1 0]
[1 0 0 1]
>>> G.random_element() in G
True

>>> F = GF(Integer(5)); MS = MatrixSpace(F,Integer(2),Integer(2))
>>> gens = [MS([[Integer(1),Integer(2)],[-Integer(1),Integer(1)]]),␣
↪MS([[Integer(1),Integer(1)],[Integer(0),Integer(1)]])]
>>> G = MatrixGroup(gens)
>>> G.random_element()  # random
[1 3]
[0 3]
>>> G.random_element() in G
True
```

subgroups()

Return a list of all the subgroups of `self`.

OUTPUT:

Each possible subgroup of `self` is contained once in the returned list. The list is in order, according to the size of the subgroups, from the trivial subgroup with one element on through up to the whole group. Conjugacy classes of subgroups are contiguous in the list.

> **Warning**
>
> For even relatively small groups this method can take a very long time to execute, or create vast amounts of output. Likely both. Its purpose is instructional, as it can be useful for studying small groups.
>
> For faster results, which still exhibit the structure of the possible subgroups, use *conjugacy_classes_subgroups()*.

EXAMPLES:

```
sage: G = groups.matrix.GL(2, 2)
sage: G.subgroups()
[Subgroup with 0 generators ()
   of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 1 generators (
 [0 1]
 [1 0]
 ) of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 1 generators (
 [1 0]
 [1 1]
 ) of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 1 generators (
 [1 1]
 [0 1]
 ) of General Linear Group of degree 2 over Finite Field of size 2,
```

(continues on next page)

(continued from previous page)

```
 Subgroup with 1 generators (
 [0 1]
 [1 1]
 ) of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 2 generators (
 [0 1]  [1 1]
 [1 1], [0 1]
 ) of General Linear Group of degree 2 over Finite Field of size 2]

sage: H = groups.matrix.Heisenberg(2)
sage: H.subgroups()
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

```
>>> from sage.all import *
>>> G = groups.matrix.GL(Integer(2), Integer(2))
>>> G.subgroups()
[Subgroup with 0 generators ()
   of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 1 generators (
 [0 1]
 [1 0]
 ) of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 1 generators (
 [1 0]
 [1 1]
 ) of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 1 generators (
 [1 1]
 [0 1]
 ) of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 1 generators (
 [0 1]
 [1 1]
 ) of General Linear Group of degree 2 over Finite Field of size 2,
 Subgroup with 2 generators (
 [0 1]  [1 1]
 [1 1], [0 1]
 ) of General Linear Group of degree 2 over Finite Field of size 2]

>>> H = groups.matrix.Heisenberg(Integer(2))
>>> H.subgroups()
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

trivial_character()

Return the trivial character of this group.

OUTPUT: a group character

EXAMPLES:

```
sage: MatrixGroup(SymmetricGroup(3)).trivial_character()                     #␣
↪needs sage.rings.number_field
Character of Matrix group over Integer Ring with 6 generators
```

```
>>> from sage.all import *
>>> MatrixGroup(SymmetricGroup(Integer(3))).trivial_character()              ␣
↪     # needs sage.rings.number_field
Character of Matrix group over Integer Ring with 6 generators
```

```
sage: GL(2,ZZ).trivial_character()
Traceback (most recent call last):
...
NotImplementedError: only implemented for finite groups
```

```
>>> from sage.all import *
>>> GL(Integer(2),ZZ).trivial_character()
Traceback (most recent call last):
...
NotImplementedError: only implemented for finite groups
```

CHAPTER

SEVEN

PARI GROUPS

See pari:polgalois for the PARI documentation of these objects.

class sage.groups.pari_group.**PariGroup**(*x*, *degree*)

Bases: object

EXAMPLES:

```
sage: PariGroup([6, -1, 2, "S3"], 3)
PARI group [6, -1, 2, S3] of degree 3
sage: R.<x> = PolynomialRing(QQ)
sage: f = x^4 - 17*x^3 - 2*x + 1
sage: G = f.galois_group(pari_group=True); G
PARI group [24, -1, 5, "S4"] of degree 4
```

```
>>> from sage.all import *
>>> PariGroup([Integer(6), -Integer(1), Integer(2), "S3"], Integer(3))
PARI group [6, -1, 2, S3] of degree 3
>>> R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> f = x**Integer(4) - Integer(17)*x**Integer(3) - Integer(2)*x + Integer(1)
>>> G = f.galois_group(pari_group=True); G
PARI group [24, -1, 5, "S4"] of degree 4
```

cardinality()

Return the order of self.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f1 = x^4 - 17*x^3 - 2*x + 1
sage: G1 = f1.galois_group(pari_group=True)
sage: G1.order()
24
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> f1 = x**Integer(4) - Integer(17)*x**Integer(3) - Integer(2)*x + Integer(1)
>>> G1 = f1.galois_group(pari_group=True)
>>> G1.order()
24
```

degree()

Return the degree of this group.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f1 = x^4 - 17*x^3 - 2*x + 1
sage: G1 = f1.galois_group(pari_group=True)
sage: G1.degree()
4
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> f1 = x**Integer(4) - Integer(17)*x**Integer(3) - Integer(2)*x + Integer(1)
>>> G1 = f1.galois_group(pari_group=True)
>>> G1.degree()
4
```

label()

Return the human readable description for this group generated by Pari.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f1 = x^4 - 17*x^3 - 2*x + 1
sage: G1 = f1.galois_group(pari_group=True)
sage: G1.label()
'S4'
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> f1 = x**Integer(4) - Integer(17)*x**Integer(3) - Integer(2)*x + Integer(1)
>>> G1 = f1.galois_group(pari_group=True)
>>> G1.label()
'S4'
```

order()

Return the order of `self`.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f1 = x^4 - 17*x^3 - 2*x + 1
sage: G1 = f1.galois_group(pari_group=True)
sage: G1.order()
24
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> f1 = x**Integer(4) - Integer(17)*x**Integer(3) - Integer(2)*x + Integer(1)
>>> G1 = f1.galois_group(pari_group=True)
>>> G1.order()
24
```

permutation_group()

Return the corresponding GAP transitive group.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^8 - x^5 + x^4 - x^3 + 1
sage: G = f.galois_group(pari_group=True)
sage: G.permutation_group()
Transitive group number 44 of degree 8
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> f = x**Integer(8) - x**Integer(5) + x**Integer(4) - x**Integer(3) +␣
↪Integer(1)
>>> G = f.galois_group(pari_group=True)
>>> G.permutation_group()
Transitive group number 44 of degree 8
```

signature()

Return 1 if contained in the alternating group, -1 otherwise.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f1 = x^4 - 17*x^3 - 2*x + 1
sage: G1 = f1.galois_group(pari_group=True)
sage: G1.signature()
-1
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> f1 = x**Integer(4) - Integer(17)*x**Integer(3) - Integer(2)*x + Integer(1)
>>> G1 = f1.galois_group(pari_group=True)
>>> G1.signature()
-1
```

transitive_number()

If the transitive label is nTk, return k.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f1 = x^4 - 17*x^3 - 2*x + 1
sage: G1 = f1.galois_group(pari_group=True)
sage: G1.transitive_number()
5
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> f1 = x**Integer(4) - Integer(17)*x**Integer(3) - Integer(2)*x + Integer(1)
>>> G1 = f1.galois_group(pari_group=True)
>>> G1.transitive_number()
5
```

CHAPTER

EIGHT

MISCELLANEOUS GENERIC FUNCTIONS

A collection of functions implementing generic algorithms in arbitrary groups, including additive and multiplicative groups.

In all cases the group operation is specified by a parameter `operation`, which is a string either one of the set of `multiplication_names` or `addition_names` specified below, or other. In the latter case, the caller must provide an identity, `inverse()` and `op()` functions.

```
multiplication_names = ('multiplication', 'times', 'product', '*')
addition_names       = ('addition', 'plus', 'sum', '+')
```

Also included are a generic function for computing multiples (or powers), and an iterator for general multiples and powers.

EXAMPLES:

Some examples in the multiplicative group of a finite field:

- Discrete logs:

```
sage: # needs sage.rings.finite_rings
sage: K = GF(3^6,'b')
sage: b = K.gen()
sage: a = b^210
sage: discrete_log(a, b, K.order() - 1)
210
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> K = GF(Integer(3)**Integer(6),'b')
>>> b = K.gen()
>>> a = b**Integer(210)
>>> discrete_log(a, b, K.order() - Integer(1))
210
```

- Linear relation finder:

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF(3^6,'a')
sage: a.multiplicative_order().factor()
2^3 * 7 * 13
sage: b = a^7
sage: c = a^13
sage: linear_relation(b,c,'*')
(13, 7)
```

(continues on next page)

(continued from previous page)

```
sage: b^13 == c^7
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> F = GF(Integer(3)**Integer(6),'a', names=('a',)); (a,) = F._first_ngens(1)
>>> a.multiplicative_order().factor()
2^3 * 7 * 13
>>> b = a**Integer(7)
>>> c = a**Integer(13)
>>> linear_relation(b,c,'*')
(13, 7)
>>> b**Integer(13) == c**Integer(7)
True
```

- Orders of elements:

```
sage: # needs sage.rings.finite_rings
sage: from sage.groups.generic import order_from_multiple, order_from_bounds
sage: k.<a> = GF(5^5)
sage: b = a^4
sage: order_from_multiple(b, 5^5 - 1, operation='*')
781
sage: order_from_bounds(b, (5^4, 5^5), operation='*')
781
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> from sage.groups.generic import order_from_multiple, order_from_bounds
>>> k = GF(Integer(5)**Integer(5), names=('a',)); (a,) = k._first_ngens(1)
>>> b = a**Integer(4)
>>> order_from_multiple(b, Integer(5)**Integer(5) - Integer(1), operation='*')
781
>>> order_from_bounds(b, (Integer(5)**Integer(4), Integer(5)**Integer(5)),␣
↪operation='*')
781
```

Some examples in the group of points of an elliptic curve over a finite field:

- Discrete logs:

```
sage: # needs sage.libs.gap sage.rings.finite_rings sage.schemes
sage: F = GF(37^2,'a')
sage: E = EllipticCurve(F,[1,1])
sage: F.<a> = GF(37^2,'a')
sage: E = EllipticCurve(F,[1,1])
sage: P = E(25*a + 16 , 15*a + 7 )
sage: P.order()
672
sage: Q = 39*P; Q
(36*a + 32 : 5*a + 12 : 1)
sage: discrete_log(Q, P, P.order(), operation='+')
39
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap sage.rings.finite_rings sage.schemes
>>> F = GF(Integer(37)**Integer(2),'a')
>>> E = EllipticCurve(F,[Integer(1),Integer(1)])
>>> F = GF(Integer(37)**Integer(2),'a', names=('a',)); (a,) = F._first_ngens(1)
>>> E = EllipticCurve(F,[Integer(1),Integer(1)])
>>> P = E(Integer(25)*a + Integer(16) , Integer(15)*a + Integer(7) )
>>> P.order()
672
>>> Q = Integer(39)*P; Q
(36*a + 32 : 5*a + 12 : 1)
>>> discrete_log(Q, P, P.order(), operation='+')
39
```

- Linear relation finder:

```
sage: # needs sage.libs.gap sage.rings.finite_rings sage.schemes
sage: F.<a> = GF(3^6,'a')
sage: E = EllipticCurve([a^5 + 2*a^3 + 2*a^2 + 2*a, a^4 + a^3 + 2*a + 1])
sage: P = E(a^5 + a^4 + a^3 + a^2 + a + 2 , 0)
sage: Q = E(2*a^3 + 2*a^2 + 2*a , a^3 + 2*a^2 + 1)
sage: linear_relation(P,Q,'+')
(1, 2)
sage: P == 2*Q
True
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap sage.rings.finite_rings sage.schemes
>>> F = GF(Integer(3)**Integer(6),'a', names=('a',)); (a,) = F._first_ngens(1)
>>> E = EllipticCurve([a**Integer(5) + Integer(2)*a**Integer(3) +␣
↪Integer(2)*a**Integer(2) + Integer(2)*a, a**Integer(4) + a**Integer(3) +␣
↪Integer(2)*a + Integer(1)])
>>> P = E(a**Integer(5) + a**Integer(4) + a**Integer(3) + a**Integer(2) + a +␣
↪Integer(2) , Integer(0))
>>> Q = E(Integer(2)*a**Integer(3) + Integer(2)*a**Integer(2) + Integer(2)*a ,␣
↪a**Integer(3) + Integer(2)*a**Integer(2) + Integer(1))
>>> linear_relation(P,Q,'+')
(1, 2)
>>> P == Integer(2)*Q
True
```

- Orders of elements:

```
sage: # needs sage.libs.gap sage.rings.finite_rings sage.schemes
sage: from sage.groups.generic import order_from_multiple, order_from_bounds
sage: k.<a> = GF(5^5)
sage: E = EllipticCurve(k,[2,4])
sage: P = E(3*a^4 + 3*a, 2*a + 1)
sage: M = E.cardinality(); M
3227
sage: plist = M.prime_factors()
sage: order_from_multiple(P, M, plist, operation='+')
3227
```

(continues on next page)

(continued from previous page)

```
sage: Q = E(0,2)
sage: order_from_multiple(Q, M, plist, operation='+')
7
sage: order_from_bounds(Q, Hasse_bounds(5^5), operation='+')
7
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap sage.rings.finite_rings sage.schemes
>>> from sage.groups.generic import order_from_multiple, order_from_bounds
>>> k = GF(Integer(5)**Integer(5), names=('a',)); (a,) = k._first_ngens(1)
>>> E = EllipticCurve(k,[Integer(2),Integer(4)])
>>> P = E(Integer(3)*a**Integer(4) + Integer(3)*a, Integer(2)*a + Integer(1))
>>> M = E.cardinality(); M
3227
>>> plist = M.prime_factors()
>>> order_from_multiple(P, M, plist, operation='+')
3227
>>> Q = E(Integer(0),Integer(2))
>>> order_from_multiple(Q, M, plist, operation='+')
7
>>> order_from_bounds(Q, Hasse_bounds(Integer(5)**Integer(5)), operation='+')
7
```

`sage.groups.generic.`**`bsgs`**`(`*a*, *b*, *bounds*, *operation='*'*, *identity=None*, *inverse=None*, *op=None*`)`

Totally generic discrete baby-step giant-step function.

Solves $na = b$ (or $a^n = b$) with $lb \leq n \leq ub$ where `bounds==(lb,ub)`, raising an error if no such n exists.

a and b must be elements of some group with given identity, inverse of `x` given by `inverse(x)`, and group operation on `x`, `y` by `op(x,y)`.

If operation is '*' or '+' then the other arguments are provided automatically; otherwise they must be provided by the caller.

INPUT:

- `a` – group element
- `b` – group element
- `bounds` – a 2-tuple of integers `(lower,upper)` with `0<=lower<=upper`
- `operation` – string: `'*'`, `'+'`, other
- `identity` – the identity element of the group
- `inverse` – function of 1 argument `x`, returning inverse of `x`
- `op` – function of 2 arguments `x`, `y` returning `x*y` in the group

OUTPUT:

An integer n such that $a^n = b$ (or $na = b$). If no such n exists, this function raises a `ValueError` exception.

Note

This is a generalization of discrete logarithm. One situation where this version is useful is to find the order of an element in a group where we only have bounds on the group order (see the elliptic curve example below).

ALGORITHM: Baby step giant step. Time and space are soft $O(\sqrt{n})$ where n is the difference between upper and lower bounds.

EXAMPLES:

```
sage: from sage.groups.generic import bsgs
sage: b = Mod(2,37);  a = b^20
sage: bsgs(b, a, (0, 36))
20

sage: p = next_prime(10^20)                                                   #␣
↪needs sage.libs.pari
sage: a = Mod(2,p); b = a^(10^25)                                             #␣
↪needs sage.libs.pari
sage: bsgs(a, b, (10^25 - 10^6, 10^25 + 10^6)) == 10^25                       #␣
↪needs sage.libs.pari
True

sage: # needs sage.rings.finite_rings
sage: K = GF(3^6,'b')
sage: a = K.gen()
sage: b = a^210
sage: bsgs(a, b, (0, K.order() - 1))
210

sage: K.<z> = CyclotomicField(230)                                            #␣
↪needs sage.rings.number_field
sage: w = z^500                                                               #␣
↪needs sage.rings.number_field
sage: bsgs(z, w, (0, 229))                                                    #␣
↪needs sage.rings.number_field
40
```

```
>>> from sage.all import *
>>> from sage.groups.generic import bsgs
>>> b = Mod(Integer(2),Integer(37));  a = b**Integer(20)
>>> bsgs(b, a, (Integer(0), Integer(36)))
20

>>> p = next_prime(Integer(10)**Integer(20))                                  ␣
↪                 # needs sage.libs.pari
>>> a = Mod(Integer(2),p); b = a**(Integer(10)**Integer(25))                  ␣
↪                          # needs sage.libs.pari
>>> bsgs(a, b, (Integer(10)**Integer(25) - Integer(10)**Integer(6),␣
↪Integer(10)**Integer(25) + Integer(10)**Integer(6))) ==␣
↪Integer(10)**Integer(25)                        # needs sage.libs.pari
True

>>> # needs sage.rings.finite_rings
>>> K = GF(Integer(3)**Integer(6),'b')
>>> a = K.gen()
>>> b = a**Integer(210)
>>> bsgs(a, b, (Integer(0), K.order() - Integer(1)))
210
```

(continues on next page)

(continued from previous page)

```
>>> K = CyclotomicField(Integer(230), names=('z',)); (z,) = K._first_ngens(1)#␣
↪needs sage.rings.number_field
>>> w = z**Integer(500)                                                      ␣
↪      # needs sage.rings.number_field
>>> bsgs(z, w, (Integer(0), Integer(229)))                                   ␣
↪              # needs sage.rings.number_field
40
```

An additive example in an elliptic curve group:

```
sage: # needs sage.rings.finite_rings sage.schemes
sage: F.<a> = GF(37^5)
sage: E = EllipticCurve(F, [1,1])
sage: P = E.lift_x(a); P
(a : 28*a^4 + 15*a^3 + 14*a^2 + 7 : 1)
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings sage.schemes
>>> F = GF(Integer(37)**Integer(5), names=('a',)); (a,) = F._first_ngens(1)
>>> E = EllipticCurve(F, [Integer(1),Integer(1)])
>>> P = E.lift_x(a); P
(a : 28*a^4 + 15*a^3 + 14*a^2 + 7 : 1)
```

This will return a multiple of the order of P:

```
sage: bsgs(P, P.parent().zero(), Hasse_bounds(F.order()), operation='+')     ␣
↪  # needs sage.rings.finite_rings sage.schemes
69327408
```

```
>>> from sage.all import *
>>> bsgs(P, P.parent().zero(), Hasse_bounds(F.order()), operation='+')
↪# needs sage.rings.finite_rings sage.schemes
69327408
```

AUTHOR:

- John Cremona (2008-03-15)

`sage.groups.generic.`**`discrete_log`**(*a*, *base*, *ord*, *bounds=None*, *operation=None*, *identity='*'*, *inverse=None*, *op=None*, *algorithm=None*, *verify='bsgs'*)

Totally generic discrete log function.

INPUT:

- `a` – group element
- `base` – group element (the base)
- `ord` – integer (multiple of order of base, or `None`)
- `bounds` – a priori bounds on the log
- `operation` – string: `'*'`, `'+'`, other
- `identity` – the group's identity

- `inverse` – function of 1 argument `x`, returning inverse of `x`
- `op` – function of 2 arguments `x`, `y`, returning `x*y` in the group
- `algorithm` – string denoting what algorithm to use for prime-order logarithms: `'bsgs'`, `'rho'`, `'lambda'`
- `verify` – boolean (default: `True`); whether to verify that output is correct before returning it.

`a` and `base` must be elements of some group with identity given by `identity`, inverse of `x` by `inverse(x)`, and group operation on `x`, `y` by `op(x,y)`.

If operation is `'*'` or `'+'`, then the other arguments are provided automatically; otherwise they must be provided by the caller.

OUTPUT:

This returns an integer n such that $b^n = a$ (or $nb = a$), assuming that `ord` is a multiple of the order of the base b. If `ord` is not specified, an attempt is made to compute it.

If no such n exists, this function raises a `ValueError` exception.

Warning

If `x` has a `log` method, it is likely to be vastly faster than using this function. E.g., if `x` is an integer modulo n, use its `log` method instead!

ALGORITHM: Pohlig-Hellman, Baby step giant step, Pollard's lambda/kangaroo, and Pollard's rho.

EXAMPLES:

```
sage: b = Mod(2,37);  a = b^20
sage: discrete_log(a, b)
20
sage: b = Mod(3,2017);  a = b^20
sage: discrete_log(a, b, bounds=(10, 100))
20

sage: # needs sage.rings.finite_rings
sage: K = GF(3^6, 'b')
sage: b = K.gen()
sage: a = b^210
sage: discrete_log(a, b, K.order() - 1)
210

sage: b = Mod(1,37);  x = Mod(2,37)
sage: discrete_log(x, b)
Traceback (most recent call last):
...
ValueError: no discrete log of 2 found to base 1
sage: b = Mod(1,997);  x = Mod(2,997)
sage: discrete_log(x, b)
Traceback (most recent call last):
...
ValueError: no discrete log of 2 found to base 1
```

```
>>> from sage.all import *
>>> b = Mod(Integer(2),Integer(37));  a = b**Integer(20)
>>> discrete_log(a, b)
20
>>> b = Mod(Integer(3),Integer(2017));  a = b**Integer(20)
>>> discrete_log(a, b, bounds=(Integer(10), Integer(100)))
20

>>> # needs sage.rings.finite_rings
>>> K = GF(Integer(3)**Integer(6), 'b')
>>> b = K.gen()
>>> a = b**Integer(210)
>>> discrete_log(a, b, K.order() - Integer(1))
210

>>> b = Mod(Integer(1),Integer(37));  x = Mod(Integer(2),Integer(37))
>>> discrete_log(x, b)
Traceback (most recent call last):
...
ValueError: no discrete log of 2 found to base 1
>>> b = Mod(Integer(1),Integer(997));  x = Mod(Integer(2),Integer(997))
>>> discrete_log(x, b)
Traceback (most recent call last):
...
ValueError: no discrete log of 2 found to base 1
```

See Issue #2356:

```
sage: F.<w> = GF(121)                                                  #␣
↪needs sage.rings.finite_rings
sage: v = w^120                                                        #␣
↪needs sage.rings.finite_rings
sage: v.log(w)                                                         #␣
↪needs sage.rings.finite_rings
0

sage: K.<z> = CyclotomicField(230)                                     #␣
↪needs sage.rings.number_field
sage: w = z^50                                                         #␣
↪needs sage.rings.number_field
sage: discrete_log(w, z)                                               #␣
↪needs sage.rings.number_field
50
```

```
>>> from sage.all import *
>>> F = GF(Integer(121), names=('w',)); (w,) = F._first_ngens(1)# needs sage.
↪rings.finite_rings
>>> v = w**Integer(120)                                                 ␣
↪      # needs sage.rings.finite_rings
>>> v.log(w)                                                          #␣
↪needs sage.rings.finite_rings
0
```

(continues on next page)

(continued from previous page)

```
>>> K = CyclotomicField(Integer(230), names=('z',)); (z,) = K._first_ngens(1)#␣
↪needs sage.rings.number_field
>>> w = z**Integer(50)                                                          ␣
↪       # needs sage.rings.number_field
>>> discrete_log(w, z)                                                        #␣
↪needs sage.rings.number_field
50
```

An example where the order is infinite: note that we must give an upper bound here:

```
sage: # needs sage.rings.number_field
sage: K.<a> = QuadraticField(23)
sage: eps = 5*a - 24         # a fundamental unit
sage: eps.multiplicative_order()
+Infinity
sage: eta = eps^100
sage: discrete_log(eta, eps, bounds=(0,1000))
100
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> K = QuadraticField(Integer(23), names=('a',)); (a,) = K._first_ngens(1)
>>> eps = Integer(5)*a - Integer(24)         # a fundamental unit
>>> eps.multiplicative_order()
+Infinity
>>> eta = eps**Integer(100)
>>> discrete_log(eta, eps, bounds=(Integer(0),Integer(1000)))
100
```

In this case we cannot detect negative powers:

```
sage: eta = eps^(-3)                                                            #␣
↪needs sage.rings.number_field
sage: discrete_log(eta,eps,bounds=(0,100))                                      #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
ValueError: no discrete log of -11515*a - 55224 found to base 5*a - 24 with␣
↪bounds (0, 100)
```

```
>>> from sage.all import *
>>> eta = eps**(-Integer(3))                                                    ␣
↪       # needs sage.rings.number_field
>>> discrete_log(eta,eps,bounds=(Integer(0),Integer(100)))                      ␣
↪               # needs sage.rings.number_field
Traceback (most recent call last):
...
ValueError: no discrete log of -11515*a - 55224 found to base 5*a - 24 with␣
↪bounds (0, 100)
```

But we can invert the base (and negate the result) instead:

```
sage: -discrete_log(eta^-1, eps, bounds=(0,100))                              #␣
↪needs sage.rings.number_field
-3
```

```
>>> from sage.all import *
>>> -discrete_log(eta**-Integer(1), eps, bounds=(Integer(0),Integer(100)))        ␣
↪                       # needs sage.rings.number_field
-3
```

An additive example: elliptic curve DLOG:

```
sage: # needs sage.libs.gap sage.rings.finite_rings sage.schemes
sage: F = GF(37^2,'a')
sage: E = EllipticCurve(F, [1,1])
sage: F.<a> = GF(37^2,'a')
sage: E = EllipticCurve(F, [1,1])
sage: P = E(25*a + 16, 15*a + 7)
sage: P.order()
672
sage: Q = 39*P; Q
(36*a + 32 : 5*a + 12 : 1)
sage: discrete_log(Q, P, P.order(), operation='+')
39
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap sage.rings.finite_rings sage.schemes
>>> F = GF(Integer(37)**Integer(2),'a')
>>> E = EllipticCurve(F, [Integer(1),Integer(1)])
>>> F = GF(Integer(37)**Integer(2),'a', names=('a',)); (a,) = F._first_ngens(1)
>>> E = EllipticCurve(F, [Integer(1),Integer(1)])
>>> P = E(Integer(25)*a + Integer(16), Integer(15)*a + Integer(7))
>>> P.order()
672
>>> Q = Integer(39)*P; Q
(36*a + 32 : 5*a + 12 : 1)
>>> discrete_log(Q, P, P.order(), operation='+')
39
```

An example of big smooth group:

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF(2^63)
sage: g = F.gen()
sage: u = g**123456789
sage: discrete_log(u,g)
123456789
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> F = GF(Integer(2)**Integer(63), names=('a',)); (a,) = F._first_ngens(1)
>>> g = F.gen()
>>> u = g**Integer(123456789)
```

(continues on next page)

(continued from previous page)

```
>>> discrete_log(u,g)
123456789
```

The above examples also work when the `'rho'` and `'lambda'` algorithms are used:

```
sage: b = Mod(2,37);  a = b^20
sage: discrete_log(a, b, algorithm='rho')
20
sage: b = Mod(3,2017);  a = b^20
sage: discrete_log(a, b, algorithm='lambda', bounds=(10, 100))
20

sage: # needs sage.rings.finite_rings
sage: K = GF(3^6,'b')
sage: b = K.gen()
sage: a = b^210
sage: discrete_log(a, b, K.order()-1, algorithm='rho')
210

sage: b = Mod(1,37);  x = Mod(2,37)
sage: discrete_log(x, b, algorithm='lambda')
Traceback (most recent call last):
...
ValueError: no discrete log of 2 found to base 1
sage: b = Mod(1,997);  x = Mod(2,997)
sage: discrete_log(x, b, algorithm='rho')
Traceback (most recent call last):
...
ValueError: no discrete log of 2 found to base 1

sage: # needs sage.libs.gap sage.rings.finite_rings sage.schemes
sage: F = GF(37^2,'a')
sage: E = EllipticCurve(F, [1,1])
sage: F.<a> = GF(37^2,'a')
sage: E = EllipticCurve(F, [1,1])
sage: P = E(25*a + 16, 15*a + 7)
sage: P.order()
672
sage: Q = 39*P; Q
(36*a + 32 : 5*a + 12 : 1)
sage: discrete_log(Q, P, P.order(), operation='+', algorithm='lambda')
39

sage: # needs sage.rings.finite_rings
sage: F.<a> = GF(2^63)
sage: g = F.gen()
sage: u = g**123456789
sage: discrete_log(u, g, algorithm='rho')
123456789
```

```
>>> from sage.all import *
>>> b = Mod(Integer(2),Integer(37));  a = b**Integer(20)
```

(continues on next page)

(continued from previous page)

```
>>> discrete_log(a, b, algorithm='rho')
20
>>> b = Mod(Integer(3),Integer(2017));   a = b**Integer(20)
>>> discrete_log(a, b, algorithm='lambda', bounds=(Integer(10), Integer(100)))
20

>>> # needs sage.rings.finite_rings
>>> K = GF(Integer(3)**Integer(6),'b')
>>> b = K.gen()
>>> a = b**Integer(210)
>>> discrete_log(a, b, K.order()-Integer(1), algorithm='rho')
210

>>> b = Mod(Integer(1),Integer(37));   x = Mod(Integer(2),Integer(37))
>>> discrete_log(x, b, algorithm='lambda')
Traceback (most recent call last):
...
ValueError: no discrete log of 2 found to base 1
>>> b = Mod(Integer(1),Integer(997));   x = Mod(Integer(2),Integer(997))
>>> discrete_log(x, b, algorithm='rho')
Traceback (most recent call last):
...
ValueError: no discrete log of 2 found to base 1

>>> # needs sage.libs.gap sage.rings.finite_rings sage.schemes
>>> F = GF(Integer(37)**Integer(2),'a')
>>> E = EllipticCurve(F, [Integer(1),Integer(1)])
>>> F = GF(Integer(37)**Integer(2),'a', names=('a',)); (a,) = F._first_ngens(1)
>>> E = EllipticCurve(F, [Integer(1),Integer(1)])
>>> P = E(Integer(25)*a + Integer(16), Integer(15)*a + Integer(7))
>>> P.order()
672
>>> Q = Integer(39)*P; Q
(36*a + 32 : 5*a + 12 : 1)
>>> discrete_log(Q, P, P.order(), operation='+', algorithm='lambda')
39

>>> # needs sage.rings.finite_rings
>>> F = GF(Integer(2)**Integer(63), names=('a',)); (a,) = F._first_ngens(1)
>>> g = F.gen()
>>> u = g**Integer(123456789)
>>> discrete_log(u, g, algorithm='rho')
123456789
```

AUTHORS:

- William Stein and David Joyner (2005-01-05)
- John Cremona (2008-02-29) rewrite using `dict()` and make generic
- Julien Grijalva (2022-08-09) rewrite to make more generic, more algorithm options, and more effective use of bounds

`sage.groups.generic.`**`discrete_log_generic`**(*a*, *base*, *ord=None*, *bounds=None*, *operation='*'*, *identity=None*, *inverse=None*, *op=None*, *algorithm='bsgs'*)

Alias for `discrete_log`.

`sage.groups.generic.`**`discrete_log_lambda`**(*a*, *base*, *bounds*, *operation='*'*, *identity=None*, *inverse=None*, *op=None*, *hash_function=<built-in function hash>*)

Pollard Lambda algorithm for computing discrete logarithms. It uses only a logarithmic amount of memory. It's useful if you have bounds on the logarithm. If you are computing logarithms in a whole finite group, you should use Pollard Rho algorithm.

INPUT:

- `a` – a group element
- `base` – a group element
- `bounds` – a couple (lb,ub) representing the range where we look for a logarithm
- `operation` – string: '+', '*' or 'other'
- `identity` – the identity element of the group
- `inverse` – function of 1 argument `x` returning inverse of `x`
- `op` – function of 2 arguments `x`, `y` returning `x*y` in the group
- `hash_function` – having an efficient hash function is critical for this algorithm

OUTPUT: integer n such that $a = base^n$ (or $a = n * base$)

ALGORITHM: Pollard Lambda, if bounds are (lb,ub) it has time complexity
O(sqrt(ub-lb)) and space complexity O(log(ub-lb))

EXAMPLES:

```
sage: F.<a> = GF(2^63)                                                    #␣
↪needs sage.rings.finite_rings
sage: discrete_log_lambda(a^1234567, a, (1200000,1250000))                #␣
↪needs sage.rings.finite_rings
1234567

sage: # needs sage.rings.finite_rings sage.schemes
sage: F.<a> = GF(37^5)
sage: E = EllipticCurve(F, [1,1])
sage: P = E.lift_x(a); P
(a : 28*a^4 + 15*a^3 + 14*a^2 + 7 : 1)
```

```
>>> from sage.all import *
>>> F = GF(Integer(2)**Integer(63), names=('a',)); (a,) = F._first_ngens(1)#␣
↪needs sage.rings.finite_rings
>>> discrete_log_lambda(a**Integer(1234567), a, (Integer(1200000),
↪Integer(1250000)))                              # needs sage.rings.finite_rings
1234567

>>> # needs sage.rings.finite_rings sage.schemes
>>> F = GF(Integer(37)**Integer(5), names=('a',)); (a,) = F._first_ngens(1)
>>> E = EllipticCurve(F, [Integer(1),Integer(1)])
>>> P = E.lift_x(a); P
(a : 28*a^4 + 15*a^3 + 14*a^2 + 7 : 1)
```

This will return a multiple of the order of P:

```
sage: discrete_log_lambda(P.parent().zero(), P, Hasse_bounds(F.order()),              ␣
↪  # needs sage.rings.finite_rings sage.schemes
....:                     operation='+')
69327408

sage: K.<a> = GF(89**5)                                                               #␣
↪needs sage.rings.finite_rings
sage: hs = lambda x: hash(x) + 15
sage: discrete_log_lambda(a**(89**3 - 3),       # long time (10s on sage.math,␣
↪2011), needs sage.rings.finite_rings
....:                     a, (89**2, 89**4), operation='*', hash_function=hs)
704966
```

```
>>> from sage.all import *
>>> discrete_log_lambda(P.parent().zero(), P, Hasse_bounds(F.order()),
↪# needs sage.rings.finite_rings sage.schemes
...                     operation='+')
69327408

>>> K = GF(Integer(89)**Integer(5), names=('a',)); (a,) = K._first_ngens(1)#␣
↪needs sage.rings.finite_rings
>>> hs = lambda x: hash(x) + Integer(15)
>>> discrete_log_lambda(a**(Integer(89)**Integer(3) - Integer(3)),       # long␣
↪time (10s on sage.math, 2011), needs sage.rings.finite_rings
...                     a, (Integer(89)**Integer(2), Integer(89)**Integer(4)),␣
↪operation='*', hash_function=hs)
704966
```

AUTHOR:

– Yann Laigle-Chapuy (2009-01-25)

`sage.groups.generic.`**`discrete_log_rho`**(*a*, *base*, *ord=None*, *operation='*'*, *identity=None*, *inverse=None*, *op=None*, *hash_function=<built-in function hash>*)

Pollard Rho algorithm for computing discrete logarithm in cyclic group of prime order. If the group order is very small it falls back to the baby step giant step algorithm.

INPUT:

- `a` – a group element
- `base` – a group element
- `ord` – the order of `base` or `None`, in this case we try to compute it
- `operation` – string (default: `'*'`); denoting whether we are in an additive group or a multiplicative one
- `identity` – the group's identity
- `inverse` – function of 1 argument `x`, returning inverse of `x`
- `op` – function of 2 arguments `x`, `y`, returning `x*y` in the group
- `hash_function` – having an efficient hash function is critical for this algorithm (see examples)

OUTPUT: integer n such that $a = base^n$ (or $a = n * base$)

ALGORITHM: Pollard rho for discrete logarithm, adapted from the article of Edlyn Teske, 'A space efficient algorithm for group structure computation'.

EXAMPLES:

```
sage: F.<a> = GF(2^13)                                              #␣
↪needs sage.rings.finite_rings
sage: g = F.gen()                                                   #␣
↪needs sage.rings.finite_rings
sage: discrete_log_rho(g^1234, g)                                   #␣
↪needs sage.rings.finite_rings
1234

sage: # needs sage.rings.finite_rings sage.schemes
sage: F.<a> = GF(37^5)
sage: E = EllipticCurve(F, [1,1])
sage: G = (3*31*2^4)*E.lift_x(a)
sage: discrete_log_rho(12345*G, G, ord=46591, operation='+')
12345
```

```
>>> from sage.all import *
>>> F = GF(Integer(2)**Integer(13), names=('a',)); (a,) = F._first_ngens(1)#␣
↪needs sage.rings.finite_rings
>>> g = F.gen()                                                   #␣
↪needs sage.rings.finite_rings
>>> discrete_log_rho(g**Integer(1234), g)                          ␣
↪       # needs sage.rings.finite_rings
1234

>>> # needs sage.rings.finite_rings sage.schemes
>>> F = GF(Integer(37)**Integer(5), names=('a',)); (a,) = F._first_ngens(1)
>>> E = EllipticCurve(F, [Integer(1),Integer(1)])
>>> G = (Integer(3)*Integer(31)*Integer(2)**Integer(4))*E.lift_x(a)
>>> discrete_log_rho(Integer(12345)*G, G, ord=Integer(46591), operation='+')
12345
```

It also works with matrices:

```
sage: A = matrix(GF(50021), [[10577, 23999, 28893],                 #␣
↪needs sage.modules sage.rings.finite_rings
....:                        [14601, 41019, 30188],
....:                        [3081, 736, 27092]])
sage: discrete_log_rho(A^1234567, A)                                #␣
↪needs sage.modules sage.rings.finite_rings
1234567
```

```
>>> from sage.all import *
>>> A = matrix(GF(Integer(50021)), [[Integer(10577), Integer(23999),␣
↪Integer(28893)],                             # needs sage.modules sage.rings.
↪finite_rings
...                        [Integer(14601), Integer(41019), Integer(30188)],
...                        [Integer(3081), Integer(736), Integer(27092)]])
>>> discrete_log_rho(A**Integer(1234567), A)                        ␣
↪       # needs sage.modules sage.rings.finite_rings
1234567
```

Beware, the order must be prime:

```
sage: I = IntegerModRing(171980)
sage: discrete_log_rho(I(2), I(3))                                          #␣
↪needs sage.libs.pari
Traceback (most recent call last):
...
ValueError: for Pollard rho algorithm the order of the group must be prime
```

```
>>> from sage.all import *
>>> I = IntegerModRing(Integer(171980))
>>> discrete_log_rho(I(Integer(2)), I(Integer(3)))                           ␣
↪               # needs sage.libs.pari
Traceback (most recent call last):
...
ValueError: for Pollard rho algorithm the order of the group must be prime
```

If it fails to find a suitable logarithm, it raises a `ValueError`:

```
sage: I = IntegerModRing(171980)
sage: discrete_log_rho(I(31002), I(15501))                                  #␣
↪needs sage.libs.pari
Traceback (most recent call last):
...
ValueError: Pollard rho algorithm failed to find a logarithm
```

```
>>> from sage.all import *
>>> I = IntegerModRing(Integer(171980))
>>> discrete_log_rho(I(Integer(31002)), I(Integer(15501)))                   ␣
↪               # needs sage.libs.pari
Traceback (most recent call last):
...
ValueError: Pollard rho algorithm failed to find a logarithm
```

The main limitation on the hash function is that we don't want to have `hash(x*y) == hash(x) + hash(y)`:

```
sage: # needs sage.libs.pari
sage: I = IntegerModRing(next_prime(2^23))
sage: def test():
....:     try:
....:          discrete_log_rho(I(123456), I(1), operation='+')
....:     except Exception:
....:          print("FAILURE")
sage: test()  # random failure
FAILURE
```

```
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> I = IntegerModRing(next_prime(Integer(2)**Integer(23)))
>>> def test():
...     try:
...          discrete_log_rho(I(Integer(123456)), I(Integer(1)), operation='+')
...     except Exception:
...          print("FAILURE")
```

(continues on next page)

(continued from previous page)

```
>>> test()  # random failure
FAILURE
```

If this happens, we can provide a better hash function:

```
sage: discrete_log_rho(I(123456), I(1), operation='+',                    #␣
↪needs sage.libs.pari
....:                  hash_function=lambda x: hash(x*x))
123456
```

```
>>> from sage.all import *
>>> discrete_log_rho(I(Integer(123456)), I(Integer(1)), operation='+',           ␣
↪              # needs sage.libs.pari
...                  hash_function=lambda x: hash(x*x))
123456
```

AUTHOR:

- Yann Laigle-Chapuy (2009-09-05)

`sage.groups.generic.`**`has_order`**`(`*P*, *n*, *operation='+'*`)`

Generic function to test if a group element P has order exactly equal to a given positive integer n.

INPUT:

- `P` – group element with respect to the specified `operation`
- `n` – positive integer, or its `Factorization`
- `operation` – string, either `'+'` (default) or `'*'`

EXAMPLES:

```
sage: from sage.groups.generic import has_order
sage: E.<P> = EllipticCurve(GF(71), [5,5])
sage: P.order()
57
sage: has_order(P, 57)
True
sage: has_order(P, factor(57))
True
sage: has_order(P, 19)
False
sage: has_order(3*P, 19)
True
sage: has_order(3*P, 57)
False
```

```
>>> from sage.all import *
>>> from sage.groups.generic import has_order
>>> E = EllipticCurve(GF(Integer(71)), [Integer(5),Integer(5)], names=('P',)); (P,
↪) = E._first_ngens(1)
>>> P.order()
57
>>> has_order(P, Integer(57))
```

(continues on next page)

(continued from previous page)

```
True
>>> has_order(P, factor(Integer(57)))
True
>>> has_order(P, Integer(19))
False
>>> has_order(Integer(3)*P, Integer(19))
True
>>> has_order(Integer(3)*P, Integer(57))
False
```

```
sage: R = Zmod(14981)
sage: g = R(321)
sage: g.multiplicative_order()
42
sage: has_order(g, 42, operation='*')
True
sage: has_order(g, factor(42), operation='*')
True
sage: has_order(g, 70, operation='*')
False
```

```
>>> from sage.all import *
>>> R = Zmod(Integer(14981))
>>> g = R(Integer(321))
>>> g.multiplicative_order()
42
>>> has_order(g, Integer(42), operation='*')
True
>>> has_order(g, factor(Integer(42)), operation='*')
True
>>> has_order(g, Integer(70), operation='*')
False
```

Note

In some cases, order *testing* can be much faster than *computing* the order using `order_from_multiple()`.

`sage.groups.generic.`**`linear_relation`**`(`*P*, *Q*, *operation='+'*, *identity=None*, *inverse=None*, *op=None*`)`

Function which solves the equation `a*P=m*Q` or `P^a=Q^m`.

Additive version: returns (a, m) with minimal $m > 0$ such that $aP = mQ$. Special case: if $\langle P\rangle$ and $\langle Q\rangle$ intersect only in $\{0\}$ then $(a, m) = (0, n)$ where n is `Q.additive_order()`.

Multiplicative version: returns (a, m) with minimal $m > 0$ such that $P^a = Q^m$. Special case: if $\langle P\rangle$ and $\langle Q\rangle$ intersect only in $\{1\}$ then $(a, m) = (0, n)$ where n is `Q.multiplicative_order()`.

ALGORITHM:

Uses the generic `bsgs()` function, and so works in general finite abelian groups.

EXAMPLES:

An additive example (in an elliptic curve group):

```
sage: # needs sage.rings.finite_rings sage.schemes
sage: F.<a> = GF(3^6,'a')
sage: E = EllipticCurve([a^5 + 2*a^3 + 2*a^2 + 2*a, a^4 + a^3 + 2*a + 1])
sage: P = E(a^5 + a^4 + a^3 + a^2 + a + 2, 0)
sage: Q = E(2*a^3 + 2*a^2 + 2*a, a^3 + 2*a^2 + 1)
sage: linear_relation(P, Q, '+')
(1, 2)
sage: P == 2*Q
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings sage.schemes
>>> F = GF(Integer(3)**Integer(6),'a', names=('a',)); (a,) = F._first_ngens(1)
>>> E = EllipticCurve([a**Integer(5) + Integer(2)*a**Integer(3) +␣
↪Integer(2)*a**Integer(2) + Integer(2)*a, a**Integer(4) + a**Integer(3) +␣
↪Integer(2)*a + Integer(1)])
>>> P = E(a**Integer(5) + a**Integer(4) + a**Integer(3) + a**Integer(2) + a +␣
↪Integer(2), Integer(0))
>>> Q = E(Integer(2)*a**Integer(3) + Integer(2)*a**Integer(2) + Integer(2)*a,␣
↪a**Integer(3) + Integer(2)*a**Integer(2) + Integer(1))
>>> linear_relation(P, Q, '+')
(1, 2)
>>> P == Integer(2)*Q
True
```

A multiplicative example (in a finite field's multiplicative group):

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF(3^6,'a')
sage: a.multiplicative_order().factor()
2^3 * 7 * 13
sage: b = a^7
sage: c = a^13
sage: linear_relation(b, c, '*')
(13, 7)
sage: b^13 == c^7
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> F = GF(Integer(3)**Integer(6),'a', names=('a',)); (a,) = F._first_ngens(1)
>>> a.multiplicative_order().factor()
2^3 * 7 * 13
>>> b = a**Integer(7)
>>> c = a**Integer(13)
>>> linear_relation(b, c, '*')
(13, 7)
>>> b**Integer(13) == c**Integer(7)
True
```

sage.groups.generic.**merge_points**(*P1*, *P2*, *operation='+'*, *identity=None*, *inverse=None*, *op=None*, *check=True*)

Return a group element whose order is the lcm of the given elements.

INPUT:

- `P1` – a pair (g_1, n_1) where g_1 is a group element of order n_1
- `P2` – a pair (g_2, n_2) where g_2 is a group element of order n_2
- `operation` – string; `'+'` (default) or `'*'` or other. If other, the following must be supplied:
  - `identity` – the identity element for the group
  - `inverse` – a function of one argument giving the inverse of a group element
  - `op` – a function of 2 arguments defining the group binary operation

OUTPUT: a pair (g_3, n_3) where g_3 has order $n_3 = \mathrm{lcm}(n_1, n_2)$

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: from sage.groups.generic import merge_points
sage: F.<a> = GF(3^6,'a')
sage: b = a^7
sage: c = a^13
sage: ob = (3^6-1)//7
sage: oc = (3^6-1)//13
sage: merge_points((b,ob), (c,oc), operation='*')
(a^4 + 2*a^3 + 2*a^2, 728)
sage: d, od = merge_points((b,ob), (c,oc), operation='*')
sage: od == d.multiplicative_order()
True
sage: od == lcm(ob, oc)
True

sage: # needs sage.libs.gap sage.rings.finite_rings sage.schemes
sage: E = EllipticCurve([a^5 + 2*a^3 + 2*a^2 + 2*a, a^4 + a^3 + 2*a + 1])
sage: P = E(2*a^5 + 2*a^4 + a^3 + 2, a^4 + a^3 + a^2 + 2*a + 2)
sage: P.order()
7
sage: Q = E(2*a^5 + 2*a^4 + 1, a^5 + 2*a^3 + 2*a + 2)
sage: Q.order()
4
sage: R, m = merge_points((P,7), (Q,4), operation='+')
sage: R.order() == m
True
sage: m == lcm(7,4)
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> from sage.groups.generic import merge_points
>>> F = GF(Integer(3)**Integer(6),'a', names=('a',)); (a,) = F._first_ngens(1)
>>> b = a**Integer(7)
>>> c = a**Integer(13)
>>> ob = (Integer(3)**Integer(6)-Integer(1))//Integer(7)
>>> oc = (Integer(3)**Integer(6)-Integer(1))//Integer(13)
>>> merge_points((b,ob), (c,oc), operation='*')
(a^4 + 2*a^3 + 2*a^2, 728)
```

(continues on next page)

(continued from previous page)

```
>>> d, od = merge_points((b,ob), (c,oc), operation='*')
>>> od == d.multiplicative_order()
True
>>> od == lcm(ob, oc)
True

>>> # needs sage.libs.gap sage.rings.finite_rings sage.schemes
>>> E = EllipticCurve([a**Integer(5) + Integer(2)*a**Integer(3) +␣
↪Integer(2)*a**Integer(2) + Integer(2)*a, a**Integer(4) + a**Integer(3) +␣
↪Integer(2)*a + Integer(1)])
>>> P = E(Integer(2)*a**Integer(5) + Integer(2)*a**Integer(4) + a**Integer(3) +␣
↪Integer(2), a**Integer(4) + a**Integer(3) + a**Integer(2) + Integer(2)*a +␣
↪Integer(2))
>>> P.order()
7
>>> Q = E(Integer(2)*a**Integer(5) + Integer(2)*a**Integer(4) + Integer(1),␣
↪a**Integer(5) + Integer(2)*a**Integer(3) + Integer(2)*a + Integer(2))
>>> Q.order()
4
>>> R, m = merge_points((P,Integer(7)), (Q,Integer(4)), operation='+')
>>> R.order() == m
True
>>> m == lcm(Integer(7),Integer(4))
True
```

sage.groups.generic.**multiple**(*a*, *n*, *operation='*'*, *identity=None*, *inverse=None*, *op=None*)

Return either na or a^n, where n is any integer and a is a Python object on which a group operation such as addition or multiplication is defined. Uses the standard binary algorithm.

INPUT: See the documentation for `discrete_logarithm()`.

EXAMPLES:

```
sage: multiple(2, 5)
32
sage: multiple(RealField()('2.5'), 4)                                      #␣
↪needs sage.rings.real_mpfr
39.0625000000000
sage: multiple(2, -3)
1/8
sage: multiple(2, 100, '+') == 100*2
True
sage: multiple(2, 100) == 2**100
True
sage: multiple(2, -100,) == 2**-100
True
sage: R.<x> = ZZ[]
sage: multiple(x, 100)
x^100
sage: multiple(x, 100, '+')
100*x
sage: multiple(x, -10)
1/x^10
```

```
>>> from sage.all import *
>>> multiple(Integer(2), Integer(5))
32
>>> multiple(RealField()('2.5'), Integer(4))                                    ␣
↪     # needs sage.rings.real_mpfr
39.0625000000000
>>> multiple(Integer(2), -Integer(3))
1/8
>>> multiple(Integer(2), Integer(100), '+') == Integer(100)*Integer(2)
True
>>> multiple(Integer(2), Integer(100)) == Integer(2)**Integer(100)
True
>>> multiple(Integer(2), -Integer(100),) == Integer(2)**-Integer(100)
True
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> multiple(x, Integer(100))
x^100
>>> multiple(x, Integer(100), '+')
100*x
>>> multiple(x, -Integer(10))
1/x^10
```

Idempotence is detected, making the following fast:

```
sage: multiple(1, 10^1000)
1

sage: # needs sage.schemes
sage: E = EllipticCurve('389a1')
sage: P = E(-1,1)
sage: multiple(P, 10, '+')
(645656132358737542773209599489/22817025904944891235367494656 :
 525532176124281192881231818644174845702936831/
↪3446581505217248068297884384990762467229696 : 1)
sage: multiple(P, -10, '+')
(645656132358737542773209599489/22817025904944891235367494656 :
 -528978757629498440949529703029165608170166527/
↪3446581505217248068297884384990762467229696 : 1)
```

```
>>> from sage.all import *
>>> multiple(Integer(1), Integer(10)**Integer(1000))
1

>>> # needs sage.schemes
>>> E = EllipticCurve('389a1')
>>> P = E(-Integer(1),Integer(1))
>>> multiple(P, Integer(10), '+')
(645656132358737542773209599489/22817025904944891235367494656 :
 525532176124281192881231818644174845702936831/
↪3446581505217248068297884384990762467229696 : 1)
>>> multiple(P, -Integer(10), '+')
(645656132358737542773209599489/22817025904944891235367494656 :
 -528978757629498440949529703029165608170166527/
```

(continues on next page)

(continued from previous page)

```
↪3446581505217248068297884384990762467229696 : 1)
```

class sage.groups.generic.**multiples**(*P*, *n*, *P0=None*, *indexed=False*, *operation='+'*, *op=None*)

Bases: `object`

Return an iterator which runs through `P0+i*P` for `i` in `range(n)`.

`P` and `P0` must be Sage objects in some group; if the operation is multiplication then the returned values are instead `P0*P**i`.

EXAMPLES:

```
sage: list(multiples(1, 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: list(multiples(1, 10, 100))
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]

sage: E = EllipticCurve('389a1')                                              #␣
↪needs sage.schemes
sage: P = E(-1,1)                                                              #␣
↪needs sage.schemes
sage: for Q in multiples(P, 5): print((Q, Q.height()/P.height()))             #␣
↪needs sage.schemes
((0 : 1 : 0), 0.000000000000000)
((-1 : 1 : 1), 1.00000000000000)
((10/9 : -35/27 : 1), 4.00000000000000)
((26/361 : -5720/6859 : 1), 9.00000000000000)
((47503/16641 : 9862190/2146689 : 1), 16.0000000000000)

sage: R.<x> = ZZ[]
sage: list(multiples(x, 5))
[0, x, 2*x, 3*x, 4*x]
sage: list(multiples(x, 5, operation='*'))
[1, x, x^2, x^3, x^4]
sage: list(multiples(x, 5, indexed=True))
[(0, 0), (1, x), (2, 2*x), (3, 3*x), (4, 4*x)]
sage: list(multiples(x, 5, indexed=True, operation='*'))
[(0, 1), (1, x), (2, x^2), (3, x^3), (4, x^4)]
sage: for i,y in multiples(x, 5, indexed=True): print("%s  times %s = %s"%(i,x,y))
0  times x = 0
1  times x = x
2  times x = 2*x
3  times x = 3*x
4  times x = 4*x

sage: for i,n in multiples(3, 5, indexed=True, operation='*'):
....:     print("3 to the power %s = %s" % (i,n))
3 to the power 0 = 1
3 to the power 1 = 3
3 to the power 2 = 9
3 to the power 3 = 27
3 to the power 4 = 81
```

```
>>> from sage.all import *
>>> list(multiples(Integer(1), Integer(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(multiples(Integer(1), Integer(10), Integer(100)))
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]

>>> E = EllipticCurve('389a1')                                              #␣
↪needs sage.schemes
>>> P = E(-Integer(1),Integer(1))                                            ␣
↪             # needs sage.schemes
>>> for Q in multiples(P, Integer(5)): print((Q, Q.height()/P.height()))     ␣
↪     # needs sage.schemes
((0 : 1 : 0), 0.000000000000000)
((-1 : 1 : 1), 1.00000000000000)
((10/9 : -35/27 : 1), 4.00000000000000)
((26/361 : -5720/6859 : 1), 9.00000000000000)
((47503/16641 : 9862190/2146689 : 1), 16.0000000000000)

>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> list(multiples(x, Integer(5)))
[0, x, 2*x, 3*x, 4*x]
>>> list(multiples(x, Integer(5), operation='*'))
[1, x, x^2, x^3, x^4]
>>> list(multiples(x, Integer(5), indexed=True))
[(0, 0), (1, x), (2, 2*x), (3, 3*x), (4, 4*x)]
>>> list(multiples(x, Integer(5), indexed=True, operation='*'))
[(0, 1), (1, x), (2, x^2), (3, x^3), (4, x^4)]
>>> for i,y in multiples(x, Integer(5), indexed=True): print("%s  times %s = %s"
↪% (i,x,y))
0  times x = 0
1  times x = x
2  times x = 2*x
3  times x = 3*x
4  times x = 4*x

>>> for i,n in multiples(Integer(3), Integer(5), indexed=True, operation='*'):
...     print("3 to the power %s = %s" % (i,n))
3 to the power 0 = 1
3 to the power 1 = 3
3 to the power 2 = 9
3 to the power 3 = 27
3 to the power 4 = 81
```

next()

Return the next item in this multiples iterator.

sage.groups.generic.**order_from_bounds**(*P*, *bounds*, *d=None*, *operation='+'*, *identity=None*, *inverse=None*, *op=None*)

Generic function to find order of a group element, given only upper and lower bounds for a multiple of the order (e.g. bounds on the order of the group of which it is an element)

INPUT:

- P – a Sage object which is a group element

- `bounds` – a 2-tuple `(lb,ub)` such that `m*P=0` (or `P**m=1`) for some `m` with `lb<=m<=ub`
- `d` – (optional) a positive integer; only `m` which are multiples of this will be considered
- `operation` – string; `'+'` (default) or `'*'` or other. If other, the following must be supplied:
  - `identity` – the identity element for the group
  - `inverse` – a function of one argument giving the inverse of a group element
  - `op` – a function of 2 arguments defining the group binary operation

Note

Typically `lb` and `ub` will be bounds on the group order, and from previous calculation we know that the group order is divisible by `d`.

EXAMPLES:

```
sage: from sage.groups.generic import order_from_bounds

sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^5)
sage: b = a^4
sage: order_from_bounds(b, (5^4, 5^5), operation='*')
781

sage: # needs sage.rings.finite_rings sage.schemes
sage: E = EllipticCurve(k, [2,4])
sage: P = E(3*a^4 + 3*a, 2*a + 1)
sage: bounds = Hasse_bounds(5^5)
sage: Q = E(0,2)
sage: order_from_bounds(Q, bounds, operation='+')
7
sage: order_from_bounds(P, bounds, 7, operation='+')
3227

sage: # needs sage.rings.number_field
sage: K.<z> = CyclotomicField(230)
sage: w = z^50
sage: order_from_bounds(w, (200, 250), operation='*')
23
```

```
>>> from sage.all import *
>>> from sage.groups.generic import order_from_bounds

>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(5), names=('a',)); (a,) = k._first_ngens(1)
>>> b = a**Integer(4)
>>> order_from_bounds(b, (Integer(5)**Integer(4), Integer(5)**Integer(5)),␣
↪operation='*')
781

>>> # needs sage.rings.finite_rings sage.schemes
>>> E = EllipticCurve(k, [Integer(2),Integer(4)])
```

(continues on next page)

(continued from previous page)

```
>>> P = E(Integer(3)*a**Integer(4) + Integer(3)*a, Integer(2)*a + Integer(1))
>>> bounds = Hasse_bounds(Integer(5)**Integer(5))
>>> Q = E(Integer(0),Integer(2))
>>> order_from_bounds(Q, bounds, operation='+')
7
>>> order_from_bounds(P, bounds, Integer(7), operation='+')
3227

>>> # needs sage.rings.number_field
>>> K = CyclotomicField(Integer(230), names=('z',)); (z,) = K._first_ngens(1)
>>> w = z**Integer(50)
>>> order_from_bounds(w, (Integer(200), Integer(250)), operation='*')
23
```

sage.groups.generic.**order_from_multiple**(*P*, *m*, *plist=None*, *factorization=None*, *check=True*, *operation='+'*, *identity=None*, *inverse=None*, *op=None*)

Generic function to find order of a group element given a multiple of its order.

See *bsgs()* for full explanation of the inputs.

INPUT:

- `P` – a Sage object which is a group element
- `m` – Sage integer which is a multiple of the order of `P`, i.e. we require that `m*P=0` (or `P**m=1`)
- `check` – a Boolean (default: `True`), indicating whether we check if `m` really is a multiple of the order
- `factorization` – the factorization of `m`, or `None` in which case this function will need to factor `m`
- `plist` – list of the prime factors of `m`, or `None`. Kept for compatibility only, prefer the use of `factorization`
- `operation` – string: `'+'` (default), `'*'` or `None`
- `identity` – the identity element of the group
- `inverse` – function of 1 argument `x`, returning inverse of `x`
- `op` – function of 2 arguments `x`, `y` returning `x*y` in the group

Note

It is more efficient for the caller to factor `m` and cache the factors for subsequent calls.

EXAMPLES:

```
sage: from sage.groups.generic import order_from_multiple

sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^5)
sage: b = a^4
sage: order_from_multiple(b, 5^5 - 1, operation='*')
781

sage: # needs sage.rings.finite_rings sage.schemes
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve(k, [2,4])
sage: P = E(3*a^4 + 3*a, 2*a + 1)
sage: M = E.cardinality(); M
3227
sage: F = M.factor()
sage: order_from_multiple(P, M, factorization=F, operation='+')
3227
sage: Q = E(0,2)
sage: order_from_multiple(Q, M, factorization=F, operation='+')
7

sage: # needs sage.rings.number_field
sage: K.<z> = CyclotomicField(230)
sage: w = z^50
sage: order_from_multiple(w, 230, operation='*')
23

sage: # needs sage.modules sage.rings.finite_rings
sage: F = GF(2^1279,'a')
sage: n = F.cardinality() - 1  # Mersenne prime
sage: order_from_multiple(F.random_element(), n,
....:                     factorization=[(n,1)], operation='*') == n
True

sage: # needs sage.rings.finite_rings
sage: K.<a> = GF(3^60)
sage: order_from_multiple(a, 3^60 - 1, operation='*', check=False)
42391158275216203514294433200
```

```
>>> from sage.all import *
>>> from sage.groups.generic import order_from_multiple

>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5)**Integer(5), names=('a',)); (a,) = k._first_ngens(1)
>>> b = a**Integer(4)
>>> order_from_multiple(b, Integer(5)**Integer(5) - Integer(1), operation='*')
781

>>> # needs sage.rings.finite_rings sage.schemes
>>> E = EllipticCurve(k, [Integer(2),Integer(4)])
>>> P = E(Integer(3)*a**Integer(4) + Integer(3)*a, Integer(2)*a + Integer(1))
>>> M = E.cardinality(); M
3227
>>> F = M.factor()
>>> order_from_multiple(P, M, factorization=F, operation='+')
3227
>>> Q = E(Integer(0),Integer(2))
>>> order_from_multiple(Q, M, factorization=F, operation='+')
7

>>> # needs sage.rings.number_field
>>> K = CyclotomicField(Integer(230), names=('z',)); (z,) = K._first_ngens(1)
```

(continues on next page)

(continued from previous page)

```
>>> w = z**Integer(50)
>>> order_from_multiple(w, Integer(230), operation='*')
23

>>> # needs sage.modules sage.rings.finite_rings
>>> F = GF(Integer(2)**Integer(1279),'a')
>>> n = F.cardinality() - Integer(1)  # Mersenne prime
>>> order_from_multiple(F.random_element(), n,
...                     factorization=[(n,Integer(1))], operation='*') == n
True

>>> # needs sage.rings.finite_rings
>>> K = GF(Integer(3)**Integer(60), names=('a',)); (a,) = K._first_ngens(1)
>>> order_from_multiple(a, Integer(3)**Integer(60) - Integer(1), operation='*',␣
→check=False)
42391158275216203514294433200
```

sage.groups.generic.**structure_description**(*G*, *latex=False*)

Return a string that tries to describe the structure of `G`.

This methods wraps GAP's `StructureDescription` method.

For full details, including the form of the returned string and the algorithm to build it, see GAP's documentation.

INPUT:

- `latex` – boolean (default: `False`); if `True`, return a LaTeX formatted string

OUTPUT: string

> **Warning**
>
> From GAP's documentation: The string returned by `StructureDescription` is **not** an isomorphism invariant: non-isomorphic groups can have the same string value, and two isomorphic groups in different representations can produce different strings.

EXAMPLES:

```
sage: # needs sage.groups
sage: G = CyclicPermutationGroup(6)
sage: G.structure_description()
'C6'
sage: G.structure_description(latex=True)
'C_{6}'
sage: G2 = G.direct_product(G, maps=False)
sage: LatexExpr(G2.structure_description(latex=True))
C_{6} \times C_{6}
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> G = CyclicPermutationGroup(Integer(6))
>>> G.structure_description()
'C6'
```

(continues on next page)

(continued from previous page)

```
>>> G.structure_description(latex=True)
'C_{6}'
>>> G2 = G.direct_product(G, maps=False)
>>> LatexExpr(G2.structure_description(latex=True))
C_{6} \times C_{6}
```

This method is mainly intended for small groups or groups with few normal subgroups. Even then there are some surprises:

```
sage: D3 = DihedralGroup(3)                                                  #␣
↪needs sage.groups
sage: D3.structure_description()                                             #␣
↪needs sage.groups
'S3'
```

```
>>> from sage.all import *
>>> D3 = DihedralGroup(Integer(3))                                            ␣
↪     # needs sage.groups
>>> D3.structure_description()                                              #␣
↪needs sage.groups
'S3'
```

We use the Sage notation for the degree of dihedral groups:

```
sage: D4 = DihedralGroup(4)                                                  #␣
↪needs sage.groups
sage: D4.structure_description()                                             #␣
↪needs sage.groups
'D4'
```

```
>>> from sage.all import *
>>> D4 = DihedralGroup(Integer(4))                                            ␣
↪     # needs sage.groups
>>> D4.structure_description()                                              #␣
↪needs sage.groups
'D4'
```

Works for finitely presented groups (Issue #17573):

```
sage: F.<x, y> = FreeGroup()                                                 #␣
↪needs sage.groups
sage: G = F / [x^2*y^-1, x^3*y^2, x*y*x^-1*y^-1]                             #␣
↪needs sage.groups
sage: G.structure_description()                                              #␣
↪needs sage.groups
'C7'
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('x', 'y',)); (x, y,) = F._first_ngens(2)# needs sage.
↪groups
>>> G = F / [x**Integer(2)*y**-Integer(1), x**Integer(3)*y**Integer(2), x*y*x**-
↪Integer(1)*y**-Integer(1)]                                   # needs sage.groups
```

(continues on next page)

(continued from previous page)

```
>>> G.structure_description()                                          #␣
↪needs sage.groups
'C7'
```

And matrix groups (Issue #17573):

```
sage: groups.matrix.GL(4,2).structure_description()                    #␣
↪needs sage.libs.gap sage.modules
'A8'
```

```
>>> from sage.all import *
>>> groups.matrix.GL(Integer(4),Integer(2)).structure_description()    ␣
↪              # needs sage.libs.gap sage.modules
'A8'
```

CHAPTER

NINE

FREE GROUPS

Free groups and finitely presented groups are implemented as a wrapper over the corresponding GAP objects.

A free group can be created by giving the number of generators, or their names. It is also possible to create indexed generators:

```
sage: G.<x,y,z> = FreeGroup();  G
Free Group on generators {x, y, z}
sage: FreeGroup(3)
Free Group on generators {x0, x1, x2}
sage: FreeGroup('a,b,c')
Free Group on generators {a, b, c}
sage: FreeGroup(3,'t')
Free Group on generators {t0, t1, t2}
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('x', 'y', 'z',)); (x, y, z,) = G._first_ngens(3);  G
Free Group on generators {x, y, z}
>>> FreeGroup(Integer(3))
Free Group on generators {x0, x1, x2}
>>> FreeGroup('a,b,c')
Free Group on generators {a, b, c}
>>> FreeGroup(Integer(3),'t')
Free Group on generators {t0, t1, t2}
```

The elements can be created by operating with the generators, or by passing a list with the indices of the letters to the group:

EXAMPLES:

```
sage: G.<a,b,c> = FreeGroup()
sage: a*b*c*a
a*b*c*a
sage: G([1,2,3,1])
a*b*c*a
sage: a * b / c * b^2
a*b*c^-1*b^2
sage: G([1,1,2,-1,-3,2])
a^2*b*a^-1*c^-1*b
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('a', 'b', 'c',)); (a, b, c,) = G._first_ngens(3)
```

(continues on next page)

(continued from previous page)

```
>>> a*b*c*a
a*b*c*a
>>> G([Integer(1),Integer(2),Integer(3),Integer(1)])
a*b*c*a
>>> a * b / c * b**Integer(2)
a*b*c^-1*b^2
>>> G([Integer(1),Integer(1),Integer(2),-Integer(1),-Integer(3),Integer(2)])
a^2*b*a^-1*c^-1*b
```

You can use call syntax to replace the generators with a set of arbitrary ring elements:

```
sage: g =  a * b / c * b^2
sage: g(1,2,3)
8/3
sage: M1 = identity_matrix(2)
sage: M2 = matrix([[1,1],[0,1]])
sage: M3 = matrix([[0,1],[1,0]])
sage: g([M1, M2, M3])
[1 3]
[1 2]
```

```
>>> from sage.all import *
>>> g =  a * b / c * b**Integer(2)
>>> g(Integer(1),Integer(2),Integer(3))
8/3
>>> M1 = identity_matrix(Integer(2))
>>> M2 = matrix([[Integer(1),Integer(1)],[Integer(0),Integer(1)]])
>>> M3 = matrix([[Integer(0),Integer(1)],[Integer(1),Integer(0)]])
>>> g([M1, M2, M3])
[1 3]
[1 2]
```

AUTHORS:

- Miguel Angel Marco Buzunariz
- Volker Braun

`sage.groups.free_group.`**`FreeGroup`**`(`*n=None*, *names='x'*, *index_set=None*, *abelian=False*, ***kwds*`)`

Construct a Free Group.

INPUT:

- `n` – integer (default: `None`); the number of generators (if not specified the `names` are counted)
- `names` – string or list/tuple/iterable of strings (default: `'x'`); the generator names or name prefix
- `index_set` – (optional) an index set for the generators; if specified then the optional keyword `abelian` can be used
- `abelian` – boolean (default: `False`); whether to construct a free abelian group or a free group

> **Note**
>
> If you want to create a free group, it is currently preferential to use `Groups().free(...)` as that does not load GAP.

EXAMPLES:

```
sage: G.<a,b> = FreeGroup();  G
Free Group on generators {a, b}
sage: H = FreeGroup('a, b')
sage: G is H
True
sage: FreeGroup(0)
Free Group on generators {}
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('a', 'b',)); (a, b,) = G._first_ngens(2);  G
Free Group on generators {a, b}
>>> H = FreeGroup('a, b')
>>> G is H
True
>>> FreeGroup(Integer(0))
Free Group on generators {}
```

The entry can be either a string with the names of the generators, or the number of generators and the prefix of the names to be given. The default prefix is `'x'`

```
sage: FreeGroup(3)
Free Group on generators {x0, x1, x2}
sage: FreeGroup(3, 'g')
Free Group on generators {g0, g1, g2}
sage: FreeGroup()
Free Group on generators {x}
```

```
>>> from sage.all import *
>>> FreeGroup(Integer(3))
Free Group on generators {x0, x1, x2}
>>> FreeGroup(Integer(3), 'g')
Free Group on generators {g0, g1, g2}
>>> FreeGroup()
Free Group on generators {x}
```

We give two examples using the `index_set` option:

```
sage: FreeGroup(index_set=ZZ)                                               #␣
↪needs sage.combinat
Free group indexed by Integer Ring
sage: FreeGroup(index_set=ZZ, abelian=True)                                 #␣
↪needs sage.combinat
Free abelian group indexed by Integer Ring
```

```
>>> from sage.all import *
>>> FreeGroup(index_set=ZZ)                                               #␣
↪needs sage.combinat
Free group indexed by Integer Ring
>>> FreeGroup(index_set=ZZ, abelian=True)                                 #␣
↪needs sage.combinat
Free abelian group indexed by Integer Ring
```

class sage.groups.free_group.**FreeGroupElement**(*parent*, *x*)

Bases: *ElementLibGAP*

A wrapper of GAP's Free Group elements.

INPUT:

- x – something that determines the group element; either a GapElement or the Tietze list (see *Tietze()*) of the group element
- parent – the parent *FreeGroup*

EXAMPLES:

```
sage: G = FreeGroup('a, b')
sage: x = G([1, 2, -1, -2])
sage: x
a*b*a^-1*b^-1
sage: y = G([2, 2, 2, 1, -2, -2, -2])
sage: y
b^3*a*b^-3
sage: x*y
a*b*a^-1*b^2*a*b^-3
sage: y*x
b^3*a*b^-3*a*b*a^-1*b^-1
sage: x^(-1)
b*a*b^-1*a^-1
sage: x == x*y*y^(-1)
True
```

```
>>> from sage.all import *
>>> G = FreeGroup('a, b')
>>> x = G([Integer(1), Integer(2), -Integer(1), -Integer(2)])
>>> x
a*b*a^-1*b^-1
>>> y = G([Integer(2), Integer(2), Integer(2), Integer(1), -Integer(2), -
↪Integer(2), -Integer(2)])
>>> y
b^3*a*b^-3
>>> x*y
a*b*a^-1*b^2*a*b^-3
>>> y*x
b^3*a*b^-3*a*b*a^-1*b^-1
>>> x**(-Integer(1))
b*a*b^-1*a^-1
>>> x == x*y*y**(-Integer(1))
True
```

Tietze()

Return the Tietze list of the element.

The Tietze list of a word is a list of integers that represent the letters in the word. A positive integer i represents the letter corresponding to the i-th generator of the group. Negative integers represent the inverses of generators.

OUTPUT: tuple of integers

EXAMPLES:

```
sage: G.<a,b> = FreeGroup()
sage: a.Tietze()
(1,)
sage: x = a^2 * b^(-3) * a^(-2)
sage: x.Tietze()
(1, 1, -2, -2, -2, -1, -1)
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('a', 'b',)); (a, b,) = G._first_ngens(2)
>>> a.Tietze()
(1,)
>>> x = a**Integer(2) * b**(-Integer(3)) * a**(-Integer(2))
>>> x.Tietze()
(1, 1, -2, -2, -2, -1, -1)
```

fox_derivative(*gen*, *im_gens=None*, *ring=None*)

Return the Fox derivative of `self` with respect to a given generator `gen` of the free group.

Let F be a free group with free generators $x_1, x_2, \ldots, x_n$. Let $j \in \{1, 2, \ldots, n\}$. Let $a_1, a_2, \ldots, a_n$ be n invertible elements of a ring A. Let $a : F \to A^\times$ be the (unique) homomorphism from F to the multiplicative group of invertible elements of A which sends each x_i to a_i. Then, we can define a map $\partial_j : F \to A$ by the requirements that

$$\partial_j(x_i) = \delta_{i,j} \qquad \text{for all indices } i \text{ and } j$$

and

$$\partial_j(uv) = \partial_j(u) + a(u)\partial_j(v) \qquad \text{for all } u, v \in F.$$

This map ∂_j is called the j-th Fox derivative on F induced by $(a_1, a_2, \ldots, a_n)$.

The most well-known case is when A is the group ring $\mathbf{Z}[F]$ of F over $\mathbf{Z}$, and when $a_i = x_i \in A$. In this case, ∂_j is simply called the j-th Fox derivative on F.

INPUT:

- `gen` – the generator with respect to which the derivative will be computed. If this is x_j, then the method will return ∂_j.
- `im_gens` – (optional) the images of the generators (given as a list or iterable). This is the list $(a_1, a_2, \ldots, a_n)$. If not provided, it defaults to $(x_1, x_2, \ldots, x_n)$ in the group ring $\mathbf{Z}[F]$.
- `ring` – (optional) the ring in which the elements of the list $(a_1, a_2, \ldots, a_n)$ lie. If not provided, this ring is inferred from these elements.

OUTPUT:

The fox derivative of `self` with respect to `gen` (induced by `im_gens`). By default, it is an element of the group algebra with integer coefficients. If `im_gens` are provided, the result lives in the algebra where `im_gens` live.

EXAMPLES:

```
sage: G = FreeGroup(5)
sage: G.inject_variables()
Defining x0, x1, x2, x3, x4
sage: (~x0*x1*x0*x2*~x0).fox_derivative(x0)
```

(continues on next page)

(continued from previous page)

```
-x0^-1 + x0^-1*x1 - x0^-1*x1*x0*x2*x0^-1
sage: (~x0*x1*x0*x2*~x0).fox_derivative(x1)
x0^-1
sage: (~x0*x1*x0*x2*~x0).fox_derivative(x2)
x0^-1*x1*x0
sage: (~x0*x1*x0*x2*~x0).fox_derivative(x3)
0
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(5))
>>> G.inject_variables()
Defining x0, x1, x2, x3, x4
>>> (~x0*x1*x0*x2*~x0).fox_derivative(x0)
-x0^-1 + x0^-1*x1 - x0^-1*x1*x0*x2*x0^-1
>>> (~x0*x1*x0*x2*~x0).fox_derivative(x1)
x0^-1
>>> (~x0*x1*x0*x2*~x0).fox_derivative(x2)
x0^-1*x1*x0
>>> (~x0*x1*x0*x2*~x0).fox_derivative(x3)
0
```

If `im_gens` is given, the images of the generators are mapped to them:

```
sage: F = FreeGroup(3)
sage: a = F([2,1,3,-1,2])
sage: a.fox_derivative(F([1]))
x1 - x1*x0*x2*x0^-1
sage: R.<t> = LaurentPolynomialRing(ZZ)
sage: a.fox_derivative(F([1]),[t,t,t])
t - t^2
sage: S.<t1,t2,t3> = LaurentPolynomialRing(ZZ)
sage: a.fox_derivative(F([1]),[t1,t2,t3])
-t2*t3 + t2
sage: R.<x,y,z> = QQ[]
sage: a.fox_derivative(F([1]),[x,y,z])
-y*z + y
sage: a.inverse().fox_derivative(F([1]),[x,y,z])
(z - 1)/(y*z)
```

```
>>> from sage.all import *
>>> F = FreeGroup(Integer(3))
>>> a = F([Integer(2),Integer(1),Integer(3),-Integer(1),Integer(2)])
>>> a.fox_derivative(F([Integer(1)]))
x1 - x1*x0*x2*x0^-1
>>> R = LaurentPolynomialRing(ZZ, names=('t',)); (t,) = R._first_ngens(1)
>>> a.fox_derivative(F([Integer(1)]),[t,t,t])
t - t^2
>>> S = LaurentPolynomialRing(ZZ, names=('t1', 't2', 't3',)); (t1, t2, t3,) =
↪S._first_ngens(3)
>>> a.fox_derivative(F([Integer(1)]),[t1,t2,t3])
-t2*t3 + t2
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
```

(continues on next page)

(continued from previous page)

```
>>> a.fox_derivative(F([Integer(1)]),[x,y,z])
-y*z + y
>>> a.inverse().fox_derivative(F([Integer(1)]),[x,y,z])
(z - 1)/(y*z)
```

The optional parameter `ring` determines the ring A:

```
sage: u = a.fox_derivative(F([1]), [1,2,3], ring=QQ)
sage: u
-4
sage: parent(u)
Rational Field
sage: u = a.fox_derivative(F([1]), [1,2,3], ring=R)
sage: u
-4
sage: parent(u)
Multivariate Polynomial Ring in x, y, z over Rational Field
```

```
>>> from sage.all import *
>>> u = a.fox_derivative(F([Integer(1)]), [Integer(1),Integer(2),Integer(3)],␣
↪ring=QQ)
>>> u
-4
>>> parent(u)
Rational Field
>>> u = a.fox_derivative(F([Integer(1)]), [Integer(1),Integer(2),Integer(3)],␣
↪ring=R)
>>> u
-4
>>> parent(u)
Multivariate Polynomial Ring in x, y, z over Rational Field
```

syllables()

Return the syllables of the word.

Consider a free group element $g = x_1^{n_1} x_2^{n_2} \cdots x_k^{n_k}$. The uniquely-determined subwords $x_i^{e_i}$ consisting only of powers of a single generator are called the syllables of g.

OUTPUT:

The tuple of syllables. Each syllable is given as a pair (x_i, e_i) consisting of a generator and a nonzero integer.

EXAMPLES:

```
sage: G.<a,b> = FreeGroup()
sage: w = a^2 * b^-1 * a^3
sage: w.syllables()
((a, 2), (b, -1), (a, 3))
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('a', 'b',)); (a, b,) = G._first_ngens(2)
>>> w = a**Integer(2) * b**-Integer(1) * a**Integer(3)
>>> w.syllables()
((a, 2), (b, -1), (a, 3))
```

class sage.groups.free_group.**FreeGroup_class**(*generator_names*, *gap_group=None*)

Bases: CachedRepresentation, *Group*, *ParentLibGAP*

A class that wraps GAP's FreeGroup.

See *FreeGroup()* for details.

Element

alias of *FreeGroupElement*

abelian_invariants()

Return the Abelian invariants of `self`.

The Abelian invariants are given by a list of integers $i_1 \dots i_j$, such that the abelianization of the group is isomorphic to

$$\mathbf{Z}/(i_1) \times \cdots \times \mathbf{Z}/(i_j)$$

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: F.abelian_invariants()
(0, 0)
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('a', 'b',)); (a, b,) = F._first_ngens(2)
>>> F.abelian_invariants()
(0, 0)
```

quotient(*relations*, ***kwds*)

Return the quotient of `self` by the normal subgroup generated by the given elements.

This quotient is a finitely presented groups with the same generators as `self`, and relations given by the elements of `relations`.

INPUT:

- `relations` – list/tuple/iterable with the elements of the free group
- further named arguments, that are passed to the constructor of a finitely presented group

OUTPUT:

A finitely presented group, with generators corresponding to the generators of the free group, and relations corresponding to the elements in `relations`.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: F.quotient([a*b^2*a, b^3])
Finitely presented group < a, b | a*b^2*a, b^3 >
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('a', 'b',)); (a, b,) = F._first_ngens(2)
>>> F.quotient([a*b**Integer(2)*a, b**Integer(3)])
Finitely presented group < a, b | a*b^2*a, b^3 >
```

Division is shorthand for *quotient()*

```
sage: F / [a*b^2*a, b^3]
Finitely presented group < a, b | a*b^2*a, b^3 >
```

```
>>> from sage.all import *
>>> F / [a*b**Integer(2)*a, b**Integer(3)]
Finitely presented group < a, b | a*b^2*a, b^3 >
```

Relations are converted to the free group, even if they are not elements of it (if possible)

```
sage: F1.<a,b,c,d> = FreeGroup()
sage: F2.<a,b> = FreeGroup()
sage: r = a*b/a
sage: r.parent()
Free Group on generators {a, b}
sage: F1/[r]
Finitely presented group < a, b, c, d | a*b*a^-1 >
```

```
>>> from sage.all import *
>>> F1 = FreeGroup(names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = F1._first_
↪ngens(4)
>>> F2 = FreeGroup(names=('a', 'b',)); (a, b,) = F2._first_ngens(2)
>>> r = a*b/a
>>> r.parent()
Free Group on generators {a, b}
>>> F1/[r]
Finitely presented group < a, b, c, d | a*b*a^-1 >
```

rank()

Return the number of generators of `self`.

Alias for `ngens()`.

OUTPUT: integer

EXAMPLES:

```
sage: G = FreeGroup('a, b'); G
Free Group on generators {a, b}
sage: G.rank()
2
sage: H = FreeGroup(3, 'x')
sage: H
Free Group on generators {x0, x1, x2}
sage: H.rank()
3
```

```
>>> from sage.all import *
>>> G = FreeGroup('a, b'); G
Free Group on generators {a, b}
>>> G.rank()
2
>>> H = FreeGroup(Integer(3), 'x')
>>> H
```

(continues on next page)

(continued from previous page)

```
Free Group on generators {x0, x1, x2}
>>> H.rank()
3
```

sage.groups.free_group.**is_FreeGroup**(*x*)

Test whether x is a *FreeGroup_class*.

INPUT:

- x – anything

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.groups.free_group import is_FreeGroup
sage: is_FreeGroup('a string')                                              #␣
↪needs sage.combinat
False
sage: is_FreeGroup(FreeGroup(0))
True
sage: is_FreeGroup(FreeGroup(index_set=ZZ))                                 #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> from sage.groups.free_group import is_FreeGroup
>>> is_FreeGroup('a string')                                              #␣
↪needs sage.combinat
False
>>> is_FreeGroup(FreeGroup(Integer(0)))
True
>>> is_FreeGroup(FreeGroup(index_set=ZZ))                                 #␣
↪needs sage.combinat
True
```

CHAPTER

TEN

FINITELY PRESENTED GROUPS

Finitely presented groups are constructed as quotients of *free_group*:

```
sage: F.<a,b,c> = FreeGroup()
sage: G = F / [a^2, b^2, c^2, a*b*c*a*b*c]
sage: G
Finitely presented group < a, b, c | a^2, b^2, c^2, (a*b*c)^2 >
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('a', 'b', 'c',)); (a, b, c,) = F._first_ngens(3)
>>> G = F / [a**Integer(2), b**Integer(2), c**Integer(2), a*b*c*a*b*c]
>>> G
Finitely presented group < a, b, c | a^2, b^2, c^2, (a*b*c)^2 >
```

One can create their elements by multiplying the generators or by specifying a Tietze list (see `Tietze()`) as in the case of free groups:

```
sage: G.gen(0) * G.gen(1)
a*b
sage: G([1,2,-1])
a*b*a^-1
sage: a.parent()
Free Group on generators {a, b, c}
sage: G.inject_variables()
Defining a, b, c
sage: a.parent()
Finitely presented group < a, b, c | a^2, b^2, c^2, (a*b*c)^2 >
```

```
>>> from sage.all import *
>>> G.gen(Integer(0)) * G.gen(Integer(1))
a*b
>>> G([Integer(1),Integer(2),-Integer(1)])
a*b*a^-1
>>> a.parent()
Free Group on generators {a, b, c}
>>> G.inject_variables()
Defining a, b, c
>>> a.parent()
Finitely presented group < a, b, c | a^2, b^2, c^2, (a*b*c)^2 >
```

Notice that, even if they are represented in the same way, the elements of a finitely presented group and the elements of the corresponding free group are not the same thing. However, they can be converted from one parent to the other:

```
sage: F.<a,b,c> = FreeGroup()
sage: G = F / [a^2,b^2,c^2,a*b*c*a*b*c]
sage: F([1])
a
sage: G([1])
a
sage: F([1]) is G([1])
False
sage: F([1]) == G([1])
False
sage: G(a*b/c)
a*b*c^-1
sage: F(G(a*b/c))
a*b*c^-1
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('a', 'b', 'c',)); (a, b, c,) = F._first_ngens(3)
>>> G = F / [a**Integer(2),b**Integer(2),c**Integer(2),a*b*c*a*b*c]
>>> F([Integer(1)])
a
>>> G([Integer(1)])
a
>>> F([Integer(1)]) is G([Integer(1)])
False
>>> F([Integer(1)]) == G([Integer(1)])
False
>>> G(a*b/c)
a*b*c^-1
>>> F(G(a*b/c))
a*b*c^-1
```

Finitely presented groups are implemented via GAP. You can use the *gap()* method to access the underlying LibGAP object:

```
sage: G = FreeGroup(2)
sage: G.inject_variables()
Defining x0, x1
sage: H = G / (x0^2, (x0*x1)^2, x1^2)
sage: H.gap()
<fp group on the generators [ x0, x1 ]>
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(2))
>>> G.inject_variables()
Defining x0, x1
>>> H = G / (x0**Integer(2), (x0*x1)**Integer(2), x1**Integer(2))
>>> H.gap()
<fp group on the generators [ x0, x1 ]>
```

This can be useful, for example, to use GAP functions that are not yet wrapped in Sage:

```
sage: H.gap().LowerCentralSeries()
[ Group(<fp, no generators known>), Group(<fp, no generators known>) ]
```

```
>>> from sage.all import *
>>> H.gap().LowerCentralSeries()
[ Group(<fp, no generators known>), Group(<fp, no generators known>) ]
```

The same holds for the group elements:

```
sage: G = FreeGroup(2)
sage: H = G / (G([1, 1]), G([2, 2, 2]), G([1, 2, -1, -2])); H
Finitely presented group < x0, x1 | x0^2, x1^3, x0*x1*x0^-1*x1^-1 >
sage: a = H([1])
sage: a
x0
sage: a.gap()
x0
sage: a.gap().Order()
2
sage: type(_)    # note that the above output is not a Sage integer
<class 'sage.libs.gap.element.GapElement_Integer'>
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(2))
>>> H = G / (G([Integer(1), Integer(1)]), G([Integer(2), Integer(2), Integer(2)]),
↪G([Integer(1), Integer(2), -Integer(1), -Integer(2)])); H
Finitely presented group < x0, x1 | x0^2, x1^3, x0*x1*x0^-1*x1^-1 >
>>> a = H([Integer(1)])
>>> a
x0
>>> a.gap()
x0
>>> a.gap().Order()
2
>>> type(_)    # note that the above output is not a Sage integer
<class 'sage.libs.gap.element.GapElement_Integer'>
```

You can use call syntax to replace the generators with a set of arbitrary ring elements. For example, take the free abelian group obtained by modding out the commutator subgroup of the free group:

```
sage: G = FreeGroup(2)
sage: G_ab = G / [G([1, 2, -1, -2])]; G_ab
Finitely presented group < x0, x1 | x0*x1*x0^-1*x1^-1 >
sage: a,b = G_ab.gens()
sage: g = a * b
sage: M1 = matrix([[1,0],[0,2]])
sage: M2 = matrix([[0,1],[1,0]])
sage: g(3, 5)
15
sage: g(M1, M1)
[1 0]
[0 4]
sage: M1*M2 == M2*M1   # matrices do not commute
False
sage: g(M1, M2)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: the values do not satisfy all relations of the group
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(2))
>>> G_ab = G / [G([Integer(1), Integer(2), -Integer(1), -Integer(2)])];  G_ab
Finitely presented group < x0, x1 | x0*x1*x0^-1*x1^-1 >
>>> a,b = G_ab.gens()
>>> g =  a * b
>>> M1 = matrix([[Integer(1),Integer(0)],[Integer(0),Integer(2)]])
>>> M2 = matrix([[Integer(0),Integer(1)],[Integer(1),Integer(0)]])
>>> g(Integer(3), Integer(5))
15
>>> g(M1, M1)
[1 0]
[0 4]
>>> M1*M2 == M2*M1   # matrices do not commute
False
>>> g(M1, M2)
Traceback (most recent call last):
...
ValueError: the values do not satisfy all relations of the group
```

Warning

Some methods are not guaranteed to finish since the word problem for finitely presented groups is, in general, undecidable. In those cases the process may run until the available memory is exhausted.

REFERENCES:

- Wikipedia article Presentation_of_a_group
- Wikipedia article Word_problem_for_groups

AUTHOR:

- Miguel Angel Marco Buzunariz

class `sage.groups.finitely_presented.`**`FinitelyPresentedGroup`**(*free_group*, *relations*, *category=None*, *libgap_fpgroup=None*)

Bases: `GroupMixinLibGAP`, `CachedRepresentation`, `Group`, `ParentLibGAP`

A class that wraps GAP's Finitely Presented Groups.

Warning

You should use `quotient()` to construct finitely presented groups as quotients of free groups. Any class inheriting this one should define `__reduce__ = CachedRepresentation.__reduce__` after importing `CachedRepresentation`.

EXAMPLES:

```
sage: G.<a,b> = FreeGroup()
sage: H = G / [a, b^3]
sage: H
Finitely presented group < a, b | a, b^3 >
sage: H.gens()
(a, b)

sage: F.<a,b> = FreeGroup('a, b')
sage: J = F / (F([1]), F([2, 2, 2]))
sage: J is H
True

sage: G = FreeGroup(2)
sage: H = G / (G([1, 1]), G([2, 2, 2]))
sage: H.gens()
(x0, x1)
sage: H.gen(0)
x0
sage: H.ngens()
2
sage: H.gap()
<fp group on the generators [ x0, x1 ]>
sage: type(_)
<class 'sage.libs.gap.element.GapElement'>
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('a', 'b',)); (a, b,) = G._first_ngens(2)
>>> H = G / [a, b**Integer(3)]
>>> H
Finitely presented group < a, b | a, b^3 >
>>> H.gens()
(a, b)

>>> F = FreeGroup('a, b', names=('a', 'b',)); (a, b,) = F._first_ngens(2)
>>> J = F / (F([Integer(1)]), F([Integer(2), Integer(2), Integer(2)]))
>>> J is H
True

>>> G = FreeGroup(Integer(2))
>>> H = G / (G([Integer(1), Integer(1)]), G([Integer(2), Integer(2), Integer(2)]))
>>> H.gens()
(x0, x1)
>>> H.gen(Integer(0))
x0
>>> H.ngens()
2
>>> H.gap()
<fp group on the generators [ x0, x1 ]>
>>> type(_)
<class 'sage.libs.gap.element.GapElement'>
```

Element

alias of *FinitelyPresentedGroupElement*

abelian_alexander_matrix(*ring=Rational Field*, *simplified=True*)

Return the Alexander matrix of the group with values in the group algebra of the abelianized.

INPUT:

- `ring` – (default: `QQ`) the base ring of the group algebra
- `simplified` – boolean (default: `False`); if set to `True` use Gauss elimination and erase rows and columns

OUTPUT:

- `A` – a matrix with coefficients in `R`
- `ideal` – an list of generators of an ideal `I` of `R = A.base_ring()` such that `R/I` is the group algebra of the abelianization of `self`

EXAMPLES:

```
sage: G.<a,b,c> = FreeGroup()
sage: H = G.quotient([a*b/a/b, a*c/a/c, c*b/c/b])
sage: A, ideal = H.abelian_alexander_matrix()
sage: A
[-f2 + 1  f1 - 1       0]
[-f3 + 1       0  f1 - 1]
[      0  f3 - 1 -f2 + 1]
sage: A.base_ring()
Multivariate Laurent Polynomial Ring in f1, f2, f3 over Rational Field
sage: ideal
[]
sage: G = FreeGroup(3)/[(2, 1, 1), (1, 2, 2, 3, 3)]
sage: A, ideal = G.abelian_alexander_matrix(simplified=True); A
[-f1^2 - f1^4 - f1^6         f1^3 + f1^6]
sage: g = FreeGroup(1) / []
sage: g.abelian_alexander_matrix()
([], [])
sage: g.abelian_alexander_matrix()[0].base_ring()
Univariate Laurent Polynomial Ring in f1 over Rational Field
sage: g = FreeGroup(0) / []
sage: A, ideal = g.abelian_alexander_matrix(); A
[]
sage: A.base_ring()
Rational Field
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('a', 'b', 'c',)); (a, b, c,) = G._first_ngens(3)
>>> H = G.quotient([a*b/a/b, a*c/a/c, c*b/c/b])
>>> A, ideal = H.abelian_alexander_matrix()
>>> A
[-f2 + 1  f1 - 1       0]
[-f3 + 1       0  f1 - 1]
[      0  f3 - 1 -f2 + 1]
>>> A.base_ring()
Multivariate Laurent Polynomial Ring in f1, f2, f3 over Rational Field
>>> ideal
[]
```

(continues on next page)

(continued from previous page)

```
>>> G = FreeGroup(Integer(3))/[(Integer(2), Integer(1), Integer(1)),␣
↪(Integer(1), Integer(2), Integer(2), Integer(3), Integer(3))]
>>> A, ideal = G.abelian_alexander_matrix(simplified=True); A
[-f1^2 - f1^4 - f1^6         f1^3 + f1^6]
>>> g = FreeGroup(Integer(1)) / []
>>> g.abelian_alexander_matrix()
([], [])
>>> g.abelian_alexander_matrix()[Integer(0)].base_ring()
Univariate Laurent Polynomial Ring in f1 over Rational Field
>>> g = FreeGroup(Integer(0)) / []
>>> A, ideal = g.abelian_alexander_matrix(); A
[]
>>> A.base_ring()
Rational Field
```

abelian_invariants()

Return the abelian invariants of `self`.

The abelian invariants are given by a list of integers $(i_1, \ldots, i_j)$, such that the abelianization of the group is isomorphic to $\mathbf{Z}/(i_1) \times \cdots \times \mathbf{Z}/(i_j)$.

EXAMPLES:

```
sage: G = FreeGroup(4, 'g')
sage: G.inject_variables()
Defining g0, g1, g2, g3
sage: H = G.quotient([g1^2, g2*g1*g2^(-1)*g1^(-1), g1*g3^(-2), g0^4])
sage: H.abelian_invariants()
(0, 4, 4)
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(4), 'g')
>>> G.inject_variables()
Defining g0, g1, g2, g3
>>> H = G.quotient([g1**Integer(2), g2*g1*g2**(-Integer(1))*g1**(-Integer(1)),
↪ g1*g3**(-Integer(2)), g0**Integer(4)])
>>> H.abelian_invariants()
(0, 4, 4)
```

ALGORITHM:

Uses GAP.

abelianization_map()

Return the abelianization map of `self`.

OUTPUT: the abelianization map of `self` as a homomorphism of finitely presented groups

EXAMPLES:

```
sage: G = FreeGroup(4, 'g')
sage: G.inject_variables(verbose=False)
sage: H = G.quotient([g1^2, g2*g1*g2^(-1)*g1^(-1), g1*g3^(-2), g0^4])
sage: H.abelianization_map()
Group morphism:
```

(continues on next page)

(continued from previous page)

```
  From: Finitely presented group < g0, g1, g2, g3 | g1^2, g2*g1*g2^-1*g1^-1,␣
↪g1*g3^-2, g0^4 >
  To:   Finitely presented group < f1, f2, f3 | f1^4, f2^-1*f1^-1*f2*f1, f2^4,
↪ f3^-1*f1^-1*f3*f1, f3^-1*f2^-1*f3*f2 >
sage: g = FreeGroup(0) / []
sage: g.abelianization_map()
Group endomorphism of Finitely presented group  <  |  >
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(4), 'g')
>>> G.inject_variables(verbose=False)
>>> H = G.quotient([g1**Integer(2), g2*g1*g2**(-Integer(1))*g1**(-Integer(1)),
↪ g1*g3**(-Integer(2)), g0**Integer(4)])
>>> H.abelianization_map()
Group morphism:
  From: Finitely presented group < g0, g1, g2, g3 | g1^2, g2*g1*g2^-1*g1^-1,␣
↪g1*g3^-2, g0^4 >
  To:   Finitely presented group < f1, f2, f3 | f1^4, f2^-1*f1^-1*f2*f1, f2^4,
↪ f3^-1*f1^-1*f3*f1, f3^-1*f2^-1*f3*f2 >
>>> g = FreeGroup(Integer(0)) / []
>>> g.abelianization_map()
Group endomorphism of Finitely presented group  <  |  >
```

abelianization_to_algebra(*ring=Rational Field*)

Return the group algebra of the abelianization of `self` together with the monomials representing the generators of `self`.

INPUT:

- `ring` – (default: `QQ`) the base ring for the group algebra of `self`

OUTPUT:

- `ab` – the abelianization of `self` as a finitely presented group with a minimal number n of generators
- `R` – a Laurent polynomial ring with n variables with base ring `ring`
- `ideal` – list of generators of an ideal `I` in `R` such that `R/I` is the group algebra of the abelianization over `ring`
- `image` – list with the images of the generators of `self` in `R/I`

EXAMPLES:

```
sage: G = FreeGroup(4, 'g')
sage: G.inject_variables()
Defining g0, g1, g2, g3
sage: H = G.quotient([g1^2, g2*g1*g2^(-1)*g1^(-1), g1*g3^(-2), g0^4])
sage: H.abelianization_to_algebra()
(Finitely presented group < f1, f2, f3 | f1^4, f2^-1*f1^-1*f2*f1, f2^4, f3^-
↪1*f1^-1*f3*f1, f3^-1*f2^-1*f3*f2 >,
 Multivariate Laurent Polynomial Ring in f1, f2, f3 over Rational Field,
 [f1^4 - 1, f2^4 - 1],
 [f1^3*f2^2, f2^2, f3, f2])
sage: g=FreeGroup(0) / []
```

(continues on next page)

(continued from previous page)

```
sage: g.abelianization_to_algebra()
(Finitely presented group  <  |  >, Rational Field, [], [])
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(4), 'g')
>>> G.inject_variables()
Defining g0, g1, g2, g3
>>> H = G.quotient([g1**Integer(2), g2*g1*g2**(-Integer(1))*g1**(-Integer(1)),
↪ g1*g3**(-Integer(2)), g0**Integer(4)])
>>> H.abelianization_to_algebra()
(Finitely presented group < f1, f2, f3 | f1^4, f2^-1*f1^-1*f2*f1, f2^4, f3^-
↪1*f1^-1*f3*f1, f3^-1*f2^-1*f3*f2 >,
 Multivariate Laurent Polynomial Ring in f1, f2, f3 over Rational Field,
 [f1^4 - 1, f2^4 - 1],
 [f1^3*f2^2, f2^2, f3, f2])
>>> g=FreeGroup(Integer(0)) / []
>>> g.abelianization_to_algebra()
(Finitely presented group  <  |  >, Rational Field, [], [])
```

alexander_matrix(*im_gens=None*)

Return the Alexander matrix of the group.

This matrix is given by the fox derivatives of the relations with respect to the generators.

- `im_gens` – (optional) the images of the generators

OUTPUT:

A matrix with coefficients in the group algebra. If `im_gens` is given, the coefficients will live in the same algebra as the given values. The result depends on the (fixed) choice of presentation.

EXAMPLES:

```
sage: G.<a,b,c> = FreeGroup()
sage: H = G.quotient([a*b/a/b, a*c/a/c, c*b/c/b])
sage: H.alexander_matrix()
[     1 - a*b*a^-1 a - a*b*a^-1*b^-1                 0]
[     1 - a*c*a^-1                 0 a - a*c*a^-1*c^-1]
[                0 c - c*b*c^-1*b^-1      1 - c*b*c^-1]
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('a', 'b', 'c',)); (a, b, c,) = G._first_ngens(3)
>>> H = G.quotient([a*b/a/b, a*c/a/c, c*b/c/b])
>>> H.alexander_matrix()
[     1 - a*b*a^-1 a - a*b*a^-1*b^-1                 0]
[     1 - a*c*a^-1                 0 a - a*c*a^-1*c^-1]
[                0 c - c*b*c^-1*b^-1      1 - c*b*c^-1]
```

If we introduce the images of the generators, we obtain the result in the corresponding algebra.

```
sage: G.<a,b,c,d,e> = FreeGroup()
sage: H = G.quotient([a*b/a/b, a*c/a/c, a*d/a/d, b*c*d/(c*d*b), b*c*d/
↪(d*b*c)])
sage: H.alexander_matrix()
```

(continues on next page)

(continued from previous page)

```
[              1 - a*b*a^-1           a - a*b*a^-1*b^-1                          ␣
↪  0                         0                          0]
[              1 - a*c*a^-1                           0           a - a*c*a^-
↪1*c^-1                         0                          0]
[              1 - a*d*a^-1                           0                          ␣
↪  0           a - a*d*a^-1*d^-1                        0]
[                         0            1 - b*c*d*b^-1   b - b*c*d*b^-1*d^-
↪1*c^-1      b*c - b*c*d*b^-1*d^-1                       0]
[                         0        1 - b*c*d*c^-1*b^-1              b -␣
↪b*c*d*c^-1 b*c - b*c*d*c^-1*b^-1*d^-1                       0]
sage: R.<t1,t2,t3,t4> = LaurentPolynomialRing(ZZ)
sage: H.alexander_matrix([t1,t2,t3,t4])
[    -t2 + 1      t1 - 1           0           0           0]
[    -t3 + 1           0      t1 - 1           0           0]
[    -t4 + 1           0           0      t1 - 1           0]
[          0  -t3*t4 + 1      t2 - 1  t2*t3 - t3           0]
[          0     -t4 + 1 -t2*t4 + t2    t2*t3 - 1           0]
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('a', 'b', 'c', 'd', 'e',)); (a, b, c, d, e,) = G._
↪first_ngens(5)
>>> H = G.quotient([a*b/a/b, a*c/a/c, a*d/a/d, b*c*d/(c*d*b), b*c*d/(d*b*c)])
>>> H.alexander_matrix()
[              1 - a*b*a^-1           a - a*b*a^-1*b^-1                          ␣
↪  0                         0                          0]
[              1 - a*c*a^-1                           0           a - a*c*a^-
↪1*c^-1                         0                          0]
[              1 - a*d*a^-1                           0                          ␣
↪  0           a - a*d*a^-1*d^-1                        0]
[                         0            1 - b*c*d*b^-1   b - b*c*d*b^-1*d^-
↪1*c^-1      b*c - b*c*d*b^-1*d^-1                       0]
[                         0        1 - b*c*d*c^-1*b^-1              b -␣
↪b*c*d*c^-1 b*c - b*c*d*c^-1*b^-1*d^-1                       0]
>>> R = LaurentPolynomialRing(ZZ, names=('t1', 't2', 't3', 't4',)); (t1, t2,␣
↪t3, t4,) = R._first_ngens(4)
>>> H.alexander_matrix([t1,t2,t3,t4])
[    -t2 + 1      t1 - 1           0           0           0]
[    -t3 + 1           0      t1 - 1           0           0]
[    -t4 + 1           0           0      t1 - 1           0]
[          0  -t3*t4 + 1      t2 - 1  t2*t3 - t3           0]
[          0     -t4 + 1 -t2*t4 + t2    t2*t3 - 1           0]
```

as_permutation_group(*limit=4096000*)

Return an isomorphic permutation group.

The generators of the resulting group correspond to the images by the isomorphism of the generators of the given group.

INPUT:

- `limit` – integer (default: 4096000); the maximal number of cosets before the computation is aborted

OUTPUT:

A Sage *PermutationGroup()*. If the number of cosets exceeds the given `limit`, a `ValueError` is returned.

EXAMPLES:

```
sage: G.<a,b> = FreeGroup()
sage: H = G / (a^2, b^3, a*b*~a*~b)
sage: H.as_permutation_group()
Permutation Group with generators [(1,2)(3,5)(4,6), (1,3,4)(2,5,6)]

sage: G.<a,b> = FreeGroup()
sage: H = G / [a^3*b]
sage: H.as_permutation_group(limit=1000)
Traceback (most recent call last):
...
ValueError: Coset enumeration exceeded limit, is the group finite?
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('a', 'b',)); (a, b,) = G._first_ngens(2)
>>> H = G / (a**Integer(2), b**Integer(3), a*b*~a*~b)
>>> H.as_permutation_group()
Permutation Group with generators [(1,2)(3,5)(4,6), (1,3,4)(2,5,6)]

>>> G = FreeGroup(names=('a', 'b',)); (a, b,) = G._first_ngens(2)
>>> H = G / [a**Integer(3)*b]
>>> H.as_permutation_group(limit=Integer(1000))
Traceback (most recent call last):
...
ValueError: Coset enumeration exceeded limit, is the group finite?
```

ALGORITHM:

Uses GAP's coset enumeration on the trivial subgroup.

> **Warning**
>
> This is in general not a decidable problem (in fact, it is not even possible to check if the group is finite or not). If the group is infinite, or too big, you should be prepared for a long computation that consumes all the memory without finishing if you do not set a sensible `limit`.

cardinality(*limit=4096000*)

Compute the cardinality of `self`.

INPUT:

- `limit` – integer (default: 4096000); the maximal number of cosets before the computation is aborted

OUTPUT: integer or `Infinity`; the number of elements in the group

EXAMPLES:

```
sage: G.<a,b> = FreeGroup('a, b')
sage: H = G / (a^2, b^3, a*b*~a*~b)
sage: H.cardinality()
6
```

(continues on next page)

(continued from previous page)

```
sage: F.<a,b,c> = FreeGroup()
sage: J = F / (F([1]), F([2, 2, 2]))
sage: J.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> G = FreeGroup('a, b', names=('a', 'b',)); (a, b,) = G._first_ngens(2)
>>> H = G / (a**Integer(2), b**Integer(3), a*b*~a*~b)
>>> H.cardinality()
6

>>> F = FreeGroup(names=('a', 'b', 'c',)); (a, b, c,) = F._first_ngens(3)
>>> J = F / (F([Integer(1)]), F([Integer(2), Integer(2), Integer(2)]))
>>> J.cardinality()
+Infinity
```

ALGORITHM:

Uses GAP.

Warning

This is in general not a decidable problem, so it is not guaranteed to give an answer. If the group is infinite, or too big, you should be prepared for a long computation that consumes all the memory without finishing if you do not set a sensible `limit`.

characteristic_varieties(*ring=Rational Field*, *matrix_ideal=None*, *groebner=False*)

Return the characteristic varieties of the group `self`.

There are several definitions of the characteristic varieties of a group G, see e.g. [CS1999a]. Let Λ be the group algebra of G/G' and $\mathbb{T}$ its associated algebraic variety (a torus). Each element $\xi \in \mathbb{T}$ defines a local system of coefficients and the k-th characteristic variety is

$$V_k(G) = \{\xi \in \mathbb{T} \mid \dim H^1(G; \xi) \geq k\}.$$

These varieties are defined by ideals in Λ.

INPUT:

- `ring` – (default: `QQ`) the base ring of the group algebra
- `groebner` – boolean (default: `False`); if set to `True` the minimal associated primes of the ideals and their groebner bases are computed; ignored if the base ring is not a field

OUTPUT:

A dictionary with keys the indices of the varieties. If `groebner` is `False` the values are the ideals defining the characteristic varieties. If it is `True`, lists for Gröbner bases for the ideal of each irreducible component, stopping when the first time a characteristic variety is empty.

EXAMPLES:

```
sage: L = [2*(i, j) + 2* (-i, -j) for i, j in ((1, 2), (2, 3), (3, 1))]
sage: G = FreeGroup(3) / L
```

(continues on next page)

(continued from previous page)

```
sage: G.characteristic_varieties(groebner=True)
{0: [(0,)],
 1: [(f1 - 1, f2 - 1, f3 - 1), (f1*f3 + 1, f2 - 1), (f1*f2 + 1, f3 - 1),␣
↪(f2*f3 + 1, f1 - 1),
     (f2*f3 + 1, f1 - f2), (f2*f3 + 1, f1 - f3), (f1*f3 + 1, f2 - f3)],
 2: [(f1 - 1, f2 - 1, f3 - 1), (f1 + 1, f2 - 1, f3 - 1), (f1 - 1, f2 - 1, f3␣
↪+ 1),
     (f3^2 + 1, f1 - f3, f2 - f3), (f1 - 1, f2 + 1, f3 - 1)],
 3: [(f1 - 1, f2 - 1, f3 - 1)],
 4: []}
sage: G = FreeGroup(2)/[2*(1,2,-1,-2)]
sage: G.characteristic_varieties()
{0: Ideal (0) of Multivariate Laurent Polynomial Ring in f1, f2 over Rational␣
↪Field,
 1: Ideal (f2 - 1, f1 - 1) of Multivariate Laurent Polynomial Ring in f1, f2␣
↪over Rational Field,
 2: Ideal (f2 - 1, f1 - 1) of Multivariate Laurent Polynomial Ring in f1, f2␣
↪over Rational Field,
 3: Ideal (1) of Multivariate Laurent Polynomial Ring in f1, f2 over Rational␣
↪Field}
sage: G.characteristic_varieties(ring=ZZ)
{0: Ideal (0) of Multivariate Laurent Polynomial Ring in f1, f2 over Integer␣
↪Ring,
 1: Ideal (2*f2 - 2, 2*f1 - 2) of Multivariate Laurent Polynomial Ring in f1,␣
↪f2 over Integer Ring,
 2: Ideal (f2 - 1, f1 - 1) of Multivariate Laurent Polynomial Ring in f1, f2␣
↪over Integer Ring,
 3: Ideal (1) of Multivariate Laurent Polynomial Ring in f1, f2 over Integer␣
↪Ring}
sage: G = FreeGroup(2)/[(1,2,1,-2,-1,-2)]
sage: G.characteristic_varieties()
{0: Ideal (0) of Univariate Laurent Polynomial Ring in f1 over Rational Field,
 1: Ideal (-1 + 2*f1 - 2*f1^2 + f1^3) of Univariate Laurent Polynomial Ring␣
↪in f1 over Rational Field,
 2: Ideal (1) of Univariate Laurent Polynomial Ring in f1 over Rational Field}
sage: G.characteristic_varieties(groebner=True)
{0: [0], 1: [-1 + f1, 1 - f1 + f1^2], 2: []}
sage: G = FreeGroup(2)/[3 * (1, ), 2 * (2, )]
sage: G.characteristic_varieties(groebner=True)
{0: [-1 + F1, 1 + F1, 1 - F1 + F1^2, 1 + F1 + F1^2], 1: [1 - F1 + F1^2],  2:␣
↪[]}
sage: G = FreeGroup(2)/[2 * (2, )]
sage: G.characteristic_varieties(groebner=True)
{0: [(f1 + 1,), (f1 - 1,)], 1: [(f1 + 1,), (f1 - 1, f2 - 1)], 2: []}
sage: G = (FreeGroup(0) / [])
sage: G.characteristic_varieties()
{0: Principal ideal (0) of Rational Field,
 1: Principal ideal (1) of Rational Field}
sage: G.characteristic_varieties(groebner=True)
{0: [(0,)], 1: [(1,)]}
```

```
>>> from sage.all import *
>>> L = [Integer(2)*(i, j) + Integer(2)* (-i, -j) for i, j in ((Integer(1),␣
↪Integer(2)), (Integer(2), Integer(3)), (Integer(3), Integer(1)))]
>>> G = FreeGroup(Integer(3)) / L
>>> G.characteristic_varieties(groebner=True)
{0: [(0,)],
 1: [(f1 - 1, f2 - 1, f3 - 1), (f1*f3 + 1, f2 - 1), (f1*f2 + 1, f3 - 1),␣
↪(f2*f3 + 1, f1 - 1),
     (f2*f3 + 1, f1 - f2), (f2*f3 + 1, f1 - f3), (f1*f3 + 1, f2 - f3)],
 2: [(f1 - 1, f2 - 1, f3 - 1), (f1 + 1, f2 - 1, f3 - 1), (f1 - 1, f2 - 1, f3␣
↪+ 1),
     (f3^2 + 1, f1 - f3, f2 - f3), (f1 - 1, f2 + 1, f3 - 1)],
 3: [(f1 - 1, f2 - 1, f3 - 1)],
 4: []}
>>> G = FreeGroup(Integer(2))/[Integer(2)*(Integer(1),Integer(2),-Integer(1),-
↪Integer(2))]
>>> G.characteristic_varieties()
{0: Ideal (0) of Multivariate Laurent Polynomial Ring in f1, f2 over Rational␣
↪Field,
 1: Ideal (f2 - 1, f1 - 1) of Multivariate Laurent Polynomial Ring in f1, f2␣
↪over Rational Field,
 2: Ideal (f2 - 1, f1 - 1) of Multivariate Laurent Polynomial Ring in f1, f2␣
↪over Rational Field,
 3: Ideal (1) of Multivariate Laurent Polynomial Ring in f1, f2 over Rational␣
↪Field}
>>> G.characteristic_varieties(ring=ZZ)
{0: Ideal (0) of Multivariate Laurent Polynomial Ring in f1, f2 over Integer␣
↪Ring,
 1: Ideal (2*f2 - 2, 2*f1 - 2) of Multivariate Laurent Polynomial Ring in f1,␣
↪f2 over Integer Ring,
 2: Ideal (f2 - 1, f1 - 1) of Multivariate Laurent Polynomial Ring in f1, f2␣
↪over Integer Ring,
 3: Ideal (1) of Multivariate Laurent Polynomial Ring in f1, f2 over Integer␣
↪Ring}
>>> G = FreeGroup(Integer(2))/[(Integer(1),Integer(2),Integer(1),-Integer(2),-
↪Integer(1),-Integer(2))]
>>> G.characteristic_varieties()
{0: Ideal (0) of Univariate Laurent Polynomial Ring in f1 over Rational Field,
 1: Ideal (-1 + 2*f1 - 2*f1^2 + f1^3) of Univariate Laurent Polynomial Ring␣
↪in f1 over Rational Field,
 2: Ideal (1) of Univariate Laurent Polynomial Ring in f1 over Rational Field}
>>> G.characteristic_varieties(groebner=True)
{0: [0], 1: [-1 + f1, 1 - f1 + f1^2], 2: []}
>>> G = FreeGroup(Integer(2))/[Integer(3) * (Integer(1), ), Integer(2) *␣
↪(Integer(2), )]
>>> G.characteristic_varieties(groebner=True)
{0: [-1 + F1, 1 + F1, 1 - F1 + F1^2, 1 + F1 + F1^2], 1: [1 - F1 + F1^2],  2:␣
↪[]}
>>> G = FreeGroup(Integer(2))/[Integer(2) * (Integer(2), )]
>>> G.characteristic_varieties(groebner=True)
{0: [(f1 + 1,), (f1 - 1,)], 1: [(f1 + 1,), (f1 - 1, f2 - 1)], 2: []}
>>> G = (FreeGroup(Integer(0)) / [])
>>> G.characteristic_varieties()
```

(continues on next page)

(continued from previous page)

```
{0: Principal ideal (0) of Rational Field,
 1: Principal ideal (1) of Rational Field}
>>> G.characteristic_varieties(groebner=True)
{0: [(0,)], 1: [(1,)]}
```

direct_product(*H*, *reduced=False*, *new_names=True*)

Return the direct product of `self` with finitely presented group `H`.

Calls GAP function `DirectProduct`, which returns the direct product of a list of groups of any representation.

From [Joh1990] (p. 45, proposition 4): If G, H are groups presented by $\langle X \mid R \rangle$ and $\langle Y \mid S \rangle$ respectively, then their direct product has the presentation $\langle X, Y \mid R, S, [X, Y] \rangle$ where $[X, Y]$ denotes the set of commutators $\{x^{-1}y^{-1}xy \mid x \in X, y \in Y\}$.

INPUT:

- `H` – a finitely presented group
- `reduced` – boolean (default: `False`); if `True`, then attempt to reduce the presentation of the product group
- `new_names` – boolean (default: `True`); if `True`, then lexicographical variable names are assigned to the generators of the group to be returned. If `False`, the group to be returned keeps the generator names of the two groups forming the direct product. Note that one cannot ask to reduce the output and ask to keep the old variable names, as they may change meaning in the output group if its presentation is reduced.

OUTPUT: the direct product of `self` with `H` as a finitely presented group

EXAMPLES:

```
sage: G = FreeGroup()
sage: C12 =  ( G / [G([1,1,1,1])] ).direct_product( G / [G([1,1,1])]); C12
Finitely presented group < a, b | a^4, b^3, a^-1*b^-1*a*b >
sage: C12.order(), C12.as_permutation_group().is_cyclic()
(12, True)
sage: klein = ( G / [G([1,1])] ).direct_product( G / [G([1,1])]); klein
Finitely presented group < a, b | a^2, b^2, a^-1*b^-1*a*b >
sage: klein.order(), klein.as_permutation_group().is_cyclic()
(4, False)
```

```
>>> from sage.all import *
>>> G = FreeGroup()
>>> C12 =  ( G / [G([Integer(1),Integer(1),Integer(1),Integer(1)])] ).direct_
↪product( G / [G([Integer(1),Integer(1),Integer(1)])]); C12
Finitely presented group < a, b | a^4, b^3, a^-1*b^-1*a*b >
>>> C12.order(), C12.as_permutation_group().is_cyclic()
(12, True)
>>> klein = ( G / [G([Integer(1),Integer(1)])] ).direct_product( G /␣
↪[G([Integer(1),Integer(1)])]); klein
Finitely presented group < a, b | a^2, b^2, a^-1*b^-1*a*b >
>>> klein.order(), klein.as_permutation_group().is_cyclic()
(4, False)
```

We can keep the variable names from `self` and `H` to examine how new relations are formed:

```
sage: F = FreeGroup("a"); G = FreeGroup("g")
sage: X = G / [G.0^12]; A = F / [F.0^6]
sage: X.direct_product(A, new_names=False)
Finitely presented group < g, a | g^12, a^6, g^-1*a^-1*g*a >
sage: A.direct_product(X, new_names=False)
Finitely presented group < a, g | a^6, g^12, a^-1*g^-1*a*g >
```

```
>>> from sage.all import *
>>> F = FreeGroup("a"); G = FreeGroup("g")
>>> X = G / [G.gen(0)**Integer(12)]; A = F / [F.gen(0)**Integer(6)]
>>> X.direct_product(A, new_names=False)
Finitely presented group < g, a | g^12, a^6, g^-1*a^-1*g*a >
>>> A.direct_product(X, new_names=False)
Finitely presented group < a, g | a^6, g^12, a^-1*g^-1*a*g >
```

Or we can attempt to reduce the output group presentation:

```
sage: F = FreeGroup("a"); G = FreeGroup("g")
sage: X = G / [G.0]; A = F / [F.0]
sage: X.direct_product(A, new_names=True)
Finitely presented group < a, b | a, b, a^-1*b^-1*a*b >
sage: X.direct_product(A, reduced=True, new_names=True)
Finitely presented group <  |  >
```

```
>>> from sage.all import *
>>> F = FreeGroup("a"); G = FreeGroup("g")
>>> X = G / [G.gen(0)]; A = F / [F.gen(0)]
>>> X.direct_product(A, new_names=True)
Finitely presented group < a, b | a, b, a^-1*b^-1*a*b >
>>> X.direct_product(A, reduced=True, new_names=True)
Finitely presented group <  |  >
```

But we cannot do both:

```
sage: K = FreeGroup(['a','b'])
sage: D = K / [K.0^5, K.1^8]
sage: D.direct_product(D, reduced=True, new_names=False)
Traceback (most recent call last):
...
ValueError: cannot reduce output and keep old variable names
```

```
>>> from sage.all import *
>>> K = FreeGroup(['a','b'])
>>> D = K / [K.gen(0)**Integer(5), K.gen(1)**Integer(8)]
>>> D.direct_product(D, reduced=True, new_names=False)
Traceback (most recent call last):
...
ValueError: cannot reduce output and keep old variable names
```

AUTHORS:

- Davis Shurbert (2013-07-20): initial version

epimorphisms(H)

Return the epimorphisms from `self` to H, up to automorphism of H.

INPUT:

- `H` – another group

EXAMPLES:

```
sage: F = FreeGroup(3)
sage: G = F / [F([1, 2, 3, 1, 2, 3]), F([1, 1, 1])]
sage: H = AlternatingGroup(3)
sage: for quo in G.epimorphisms(H):
....:     for a in G.gens():
....:         print(a, "|-->", quo(a))
....:     print("-----")
x0 |--> ()
x1 |--> (1,3,2)
x2 |--> (1,2,3)
-----
x0 |--> (1,3,2)
x1 |--> ()
x2 |--> (1,2,3)
-----
x0 |--> (1,3,2)
x1 |--> (1,2,3)
x2 |--> ()
-----
x0 |--> (1,2,3)
x1 |--> (1,2,3)
x2 |--> (1,2,3)
-----
```

```
>>> from sage.all import *
>>> F = FreeGroup(Integer(3))
>>> G = F / [F([Integer(1), Integer(2), Integer(3), Integer(1), Integer(2),␣
↪Integer(3)]), F([Integer(1), Integer(1), Integer(1)])]
>>> H = AlternatingGroup(Integer(3))
>>> for quo in G.epimorphisms(H):
...     for a in G.gens():
...         print(a, "|-->", quo(a))
...     print("-----")
x0 |--> ()
x1 |--> (1,3,2)
x2 |--> (1,2,3)
-----
x0 |--> (1,3,2)
x1 |--> ()
x2 |--> (1,2,3)
-----
x0 |--> (1,3,2)
x1 |--> (1,2,3)
x2 |--> ()
-----
```

(continues on next page)

(continued from previous page)

```
x0 |--> (1,2,3)
x1 |--> (1,2,3)
x2 |--> (1,2,3)
-----
```

ALGORITHM:

Uses libgap's GQuotients function.

free_group()

Return the free group (without relations).

OUTPUT: a *FreeGroup()*

EXAMPLES:

```
sage: G.<a,b,c> = FreeGroup()
sage: H = G / (a^2, b^3, a*b*~a*~b)
sage: H.free_group()
Free Group on generators {a, b, c}
sage: H.free_group() is G
True
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('a', 'b', 'c',)); (a, b, c,) = G._first_ngens(3)
>>> H = G / (a**Integer(2), b**Integer(3), a*b*~a*~b)
>>> H.free_group()
Free Group on generators {a, b, c}
>>> H.free_group() is G
True
```

order(*limit=4096000*)

Compute the cardinality of `self`.

INPUT:

- `limit` – integer (default: 4096000); the maximal number of cosets before the computation is aborted

OUTPUT: integer or `Infinity`; the number of elements in the group

EXAMPLES:

```
sage: G.<a,b> = FreeGroup('a, b')
sage: H = G / (a^2, b^3, a*b*~a*~b)
sage: H.cardinality()
6

sage: F.<a,b,c> = FreeGroup()
sage: J = F / (F([1]), F([2, 2, 2]))
sage: J.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> G = FreeGroup('a, b', names=('a', 'b',)); (a, b,) = G._first_ngens(2)
>>> H = G / (a**Integer(2), b**Integer(3), a*b*~a*~b)
```

(continues on next page)

(continued from previous page)

```
>>> H.cardinality()
6

>>> F = FreeGroup(names=('a', 'b', 'c',)); (a, b, c,) = F._first_ngens(3)
>>> J = F / (F([Integer(1)]), F([Integer(2), Integer(2), Integer(2)]))
>>> J.cardinality()
+Infinity
```

ALGORITHM:

Uses GAP.

Warning

This is in general not a decidable problem, so it is not guaranteed to give an answer. If the group is infinite, or too big, you should be prepared for a long computation that consumes all the memory without finishing if you do not set a sensible `limit`.

relations()

Return the relations of the group.

OUTPUT: the relations as a tuple of elements of *free_group()*

EXAMPLES:

```
sage: F = FreeGroup(5, 'x')
sage: F.inject_variables()
Defining x0, x1, x2, x3, x4
sage: G = F.quotient([x0*x2, x3*x1*x3, x2*x1*x2])
sage: G.relations()
(x0*x2, x3*x1*x3, x2*x1*x2)
sage: all(rel in F for rel in G.relations())
True
```

```
>>> from sage.all import *
>>> F = FreeGroup(Integer(5), 'x')
>>> F.inject_variables()
Defining x0, x1, x2, x3, x4
>>> G = F.quotient([x0*x2, x3*x1*x3, x2*x1*x2])
>>> G.relations()
(x0*x2, x3*x1*x3, x2*x1*x2)
>>> all(rel in F for rel in G.relations())
True
```

rewriting_system()

Return the rewriting system corresponding to the finitely presented group. This rewriting system can be used to reduce words with respect to the relations.

If the rewriting system is transformed into a confluent one, the reduction process will give as a result the (unique) reduced form of an element.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G = F / [a^2,b^3,(a*b/a)^3,b*a*b*a]
sage: k = G.rewriting_system()
sage: k
Rewriting system of Finitely presented group < a, b | a^2, b^3, a*b^3*a^-1,␣
↪b*a*b*a >
with rules:
    a^2    --->    1
    b^3    --->    1
    b*a*b*a    --->    1
    a*b^3*a^-1    --->    1

sage: G([1,1,2,2,2])
a^2*b^3
sage: k.reduce(G([1,1,2,2,2]))
1
sage: k.reduce(G([2,2,1]))
b^2*a
sage: k.make_confluent()
sage: k.reduce(G([2,2,1]))
a*b
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('a', 'b',)); (a, b,) = F._first_ngens(2)
>>> G = F / [a**Integer(2),b**Integer(3),(a*b/a)**Integer(3),b*a*b*a]
>>> k = G.rewriting_system()
>>> k
Rewriting system of Finitely presented group < a, b | a^2, b^3, a*b^3*a^-1,␣
↪b*a*b*a >
with rules:
    a^2    --->    1
    b^3    --->    1
    b*a*b*a    --->    1
    a*b^3*a^-1    --->    1

>>> G([Integer(1),Integer(1),Integer(2),Integer(2),Integer(2)])
a^2*b^3
>>> k.reduce(G([Integer(1),Integer(1),Integer(2),Integer(2),Integer(2)]))
1
>>> k.reduce(G([Integer(2),Integer(2),Integer(1)]))
b^2*a
>>> k.make_confluent()
>>> k.reduce(G([Integer(2),Integer(2),Integer(1)]))
a*b
```

semidirect_product (*H*, *hom*, *check=True*, *reduced=False*)

The semidirect product of `self` with `H` via `hom`.

If there exists a homomorphism ϕ from a group G to the automorphism group of a group H, then we can define the semidirect product of G with H via ϕ as the Cartesian product of G and H with the operation

$$(g_1, h_1)(g_2, h_2) = (g_1 g_2, \phi(g_2)(h_1) h_2).$$

INPUT:

- `H` – finitely presented group which is implicitly acted on by `self` and can be naturally embedded as a normal subgroup of the semidirect product
- `hom` – homomorphism from `self` to the automorphism group of `H`. Given as a pair, with generators of `self` in the first slot and the images of the corresponding generators in the second. These images must be automorphisms of `H`, given again as a pair of generators and images.
- `check` – boolean (default: `True`); if `False` the defining homomorphism and automorphism images are not tested for validity. This test can be costly with large groups, so it can be bypassed if the user is confident that his morphisms are valid.
- `reduced` – boolean (default: `False`); if `True` then the method attempts to reduce the presentation of the output group

OUTPUT:

The semidirect product of `self` with `H` via `hom` as a finitely presented group. See *PermutationGroup_generic.semidirect_product* for a more in depth explanation of a semidirect product.

AUTHORS:

- Davis Shurbert (8-1-2013)

EXAMPLES:

Group of order 12 as two isomorphic semidirect products:

```
sage: D4 = groups.presentation.Dihedral(4)
sage: C3 = groups.presentation.Cyclic(3)
sage: alpha1 = ([C3.gen(0)],[C3.gen(0)])
sage: alpha2 = ([C3.gen(0)],[C3([1,1])])
sage: S1 = D4.semidirect_product(C3, ([D4.gen(1), D4.gen(0)],[alpha1,alpha2]))
sage: C2 = groups.presentation.Cyclic(2)
sage: Q = groups.presentation.DiCyclic(3)
sage: a = Q([1]); b = Q([-2])
sage: alpha = (Q.gens(), [a,b])
sage: S2 = C2.semidirect_product(Q, ([C2.0],[alpha]))
sage: S1.is_isomorphic(S2)
True
```

```
>>> from sage.all import *
>>> D4 = groups.presentation.Dihedral(Integer(4))
>>> C3 = groups.presentation.Cyclic(Integer(3))
>>> alpha1 = ([C3.gen(Integer(0))],[C3.gen(Integer(0))])
>>> alpha2 = ([C3.gen(Integer(0))],[C3([Integer(1),Integer(1)])])
>>> S1 = D4.semidirect_product(C3, ([D4.gen(Integer(1)), D4.gen(Integer(0))],
↪[alpha1,alpha2]))
>>> C2 = groups.presentation.Cyclic(Integer(2))
>>> Q = groups.presentation.DiCyclic(Integer(3))
>>> a = Q([Integer(1)]); b = Q([-Integer(2)])
>>> alpha = (Q.gens(), [a,b])
>>> S2 = C2.semidirect_product(Q, ([C2.gen(0)],[alpha]))
>>> S1.is_isomorphic(S2)
True
```

Dihedral groups can be constructed as semidirect products of cyclic groups:

```
sage: C2 = groups.presentation.Cyclic(2)
sage: C8 = groups.presentation.Cyclic(8)
sage: hom = (C2.gens(), [ ([C8([1])], [C8([-1])]) ])
sage: D = C2.semidirect_product(C8, hom)
sage: D.as_permutation_group().is_isomorphic(DihedralGroup(8))
True
```

```
>>> from sage.all import *
>>> C2 = groups.presentation.Cyclic(Integer(2))
>>> C8 = groups.presentation.Cyclic(Integer(8))
>>> hom = (C2.gens(), [ ([C8([Integer(1)])], [C8([-Integer(1)])]) ])
>>> D = C2.semidirect_product(C8, hom)
>>> D.as_permutation_group().is_isomorphic(DihedralGroup(Integer(8)))
True
```

You can attempt to reduce the presentation of the output group:

```
sage: D = C2.semidirect_product(C8, hom); D
Finitely presented group < a, b | a^2, b^8, a^-1*b*a*b >
sage: D = C2.semidirect_product(C8, hom, reduced=True); D
Finitely presented group < a, b | a^2, a*b*a*b, b^8 >

sage: C3 = groups.presentation.Cyclic(3)
sage: C4 = groups.presentation.Cyclic(4)
sage: hom = (C3.gens(), [(C4.gens(), C4.gens())])
sage: C3.semidirect_product(C4, hom)
Finitely presented group < a, b | a^3, b^4, a^-1*b*a*b^-1 >
sage: D = C3.semidirect_product(C4, hom, reduced=True); D
Finitely presented group < a, b | a^3, b^4, a^-1*b*a*b^-1 >
sage: D.as_permutation_group().is_cyclic()
True
```

```
>>> from sage.all import *
>>> D = C2.semidirect_product(C8, hom); D
Finitely presented group < a, b | a^2, b^8, a^-1*b*a*b >
>>> D = C2.semidirect_product(C8, hom, reduced=True); D
Finitely presented group < a, b | a^2, a*b*a*b, b^8 >

>>> C3 = groups.presentation.Cyclic(Integer(3))
>>> C4 = groups.presentation.Cyclic(Integer(4))
>>> hom = (C3.gens(), [(C4.gens(), C4.gens())])
>>> C3.semidirect_product(C4, hom)
Finitely presented group < a, b | a^3, b^4, a^-1*b*a*b^-1 >
>>> D = C3.semidirect_product(C4, hom, reduced=True); D
Finitely presented group < a, b | a^3, b^4, a^-1*b*a*b^-1 >
>>> D.as_permutation_group().is_cyclic()
True
```

You can turn off the checks for the validity of the input morphisms. This check is expensive but behavior is unpredictable if inputs are invalid and are not caught by these tests:

```
sage: C5 = groups.presentation.Cyclic(5)
sage: C12 = groups.presentation.Cyclic(12)
```

(continues on next page)

(continued from previous page)

```
sage: hom = (C5.gens(), [(C12.gens(), C12.gens())])
sage: sp = C5.semidirect_product(C12, hom, check=False); sp
Finitely presented group < a, b | a^5, b^12, a^-1*b*a*b^-1 >
sage: sp.as_permutation_group().is_cyclic(), sp.order()
(True, 60)
```

```
>>> from sage.all import *
>>> C5 = groups.presentation.Cyclic(Integer(5))
>>> C12 = groups.presentation.Cyclic(Integer(12))
>>> hom = (C5.gens(), [(C12.gens(), C12.gens())])
>>> sp = C5.semidirect_product(C12, hom, check=False); sp
Finitely presented group < a, b | a^5, b^12, a^-1*b*a*b^-1 >
>>> sp.as_permutation_group().is_cyclic(), sp.order()
(True, 60)
```

simplification_isomorphism()

Return an isomorphism from `self` to a finitely presented group with a (hopefully) simpler presentation.

EXAMPLES:

```
sage: G.<a,b,c> = FreeGroup()
sage: H = G / [a*b*c, a*b^2, c*b/c^2]
sage: I = H.simplification_isomorphism()
sage: I
Group morphism:
  From: Finitely presented group < a, b, c | a*b*c, a*b^2, c*b*c^-2 >
  To:   Finitely presented group < b |  >
sage: I(a)
b^-2
sage: I(b)
b
sage: I(c)
b
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('a', 'b', 'c',)); (a, b, c,) = G._first_ngens(3)
>>> H = G / [a*b*c, a*b**Integer(2), c*b/c**Integer(2)]
>>> I = H.simplification_isomorphism()
>>> I
Group morphism:
  From: Finitely presented group < a, b, c | a*b*c, a*b^2, c*b*c^-2 >
  To:   Finitely presented group < b |  >
>>> I(a)
b^-2
>>> I(b)
b
>>> I(c)
b
```

ALGORITHM:

Uses GAP.

simplified()

Return an isomorphic group with a (hopefully) simpler presentation.

OUTPUT:

A new finitely presented group. Use *simplification_isomorphism()* if you want to know the isomorphism.

EXAMPLES:

```
sage: G.<x,y> = FreeGroup()
sage: H = G /  [x ^5, y ^4, y*x*y^3*x ^3]
sage: H
Finitely presented group < x, y | x^5, y^4, y*x*y^3*x^3 >
sage: H.simplified()
Finitely presented group < x, y | y^4, y*x*y^-1*x^-2, x^5 >
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('x', 'y',)); (x, y,) = G._first_ngens(2)
>>> H = G /  [x **Integer(5), y **Integer(4), y*x*y**Integer(3)*x
→**Integer(3)]
>>> H
Finitely presented group < x, y | x^5, y^4, y*x*y^3*x^3 >
>>> H.simplified()
Finitely presented group < x, y | y^4, y*x*y^-1*x^-2, x^5 >
```

A more complicate example:

```
sage: G.<e0, e1, e2, e3, e4, e5, e6, e7, e8, e9> = FreeGroup()
sage: rels = [e6, e5, e3, e9, e4*e7^-1*e6, e9*e7^-1*e0,
....:          e0*e1^-1*e2, e5*e1^-1*e8, e4*e3^-1*e8, e2]
sage: H = G.quotient(rels);  H
Finitely presented group < e0, e1, e2, e3, e4, e5, e6, e7, e8, e9 |
e6, e5, e3, e9, e4*e7^-1*e6, e9*e7^-1*e0, e0*e1^-1*e2, e5*e1^-1*e8, e4*e3^-
→1*e8, e2 >
sage: H.simplified()
Finitely presented group < e0 | e0^2 >
```

```
>>> from sage.all import *
>>> G = FreeGroup(names=('e0', 'e1', 'e2', 'e3', 'e4', 'e5', 'e6', 'e7', 'e8',
→ 'e9',)); (e0, e1, e2, e3, e4, e5, e6, e7, e8, e9,) = G._first_ngens(10)
>>> rels = [e6, e5, e3, e9, e4*e7**-Integer(1)*e6, e9*e7**-Integer(1)*e0,
...          e0*e1**-Integer(1)*e2, e5*e1**-Integer(1)*e8, e4*e3**-
→Integer(1)*e8, e2]
>>> H = G.quotient(rels);  H
Finitely presented group < e0, e1, e2, e3, e4, e5, e6, e7, e8, e9 |
e6, e5, e3, e9, e4*e7^-1*e6, e9*e7^-1*e0, e0*e1^-1*e2, e5*e1^-1*e8, e4*e3^-
→1*e8, e2 >
>>> H.simplified()
Finitely presented group < e0 | e0^2 >
```

sorted_presentation()

Return the same presentation with the relations sorted to ensure equality.

OUTPUT: a new finitely presented group with the relations sorted

EXAMPLES:

```
sage: G = FreeGroup(2) / [(1, 2, -1, -2), ()]; G
Finitely presented group < x0, x1 | x0*x1*x0^-1*x1^-1, 1 >
sage: G.sorted_presentation()
Finitely presented group < x0, x1 | 1, x1^-1*x0^-1*x1*x0 >
```

```
>>> from sage.all import *
>>> G = FreeGroup(Integer(2)) / [(Integer(1), Integer(2), -Integer(1), -
↪Integer(2)), ()]; G
Finitely presented group < x0, x1 | x0*x1*x0^-1*x1^-1, 1 >
>>> G.sorted_presentation()
Finitely presented group < x0, x1 | 1, x1^-1*x0^-1*x1*x0 >
```

structure_description(*G*, *latex=False*)

Return a string that tries to describe the structure of `G`.

This methods wraps GAP's `StructureDescription` method.

For full details, including the form of the returned string and the algorithm to build it, see GAP's documentation.

INPUT:

- `latex` – boolean (default: `False`); if `True`, return a LaTeX formatted string

OUTPUT: string

Warning

From GAP's documentation: The string returned by `StructureDescription` is **not** an isomorphism invariant: non-isomorphic groups can have the same string value, and two isomorphic groups in different representations can produce different strings.

EXAMPLES:

```
sage: # needs sage.groups
sage: G = CyclicPermutationGroup(6)
sage: G.structure_description()
'C6'
sage: G.structure_description(latex=True)
'C_{6}'
sage: G2 = G.direct_product(G, maps=False)
sage: LatexExpr(G2.structure_description(latex=True))
C_{6} \times C_{6}
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> G = CyclicPermutationGroup(Integer(6))
>>> G.structure_description()
'C6'
>>> G.structure_description(latex=True)
'C_{6}'
>>> G2 = G.direct_product(G, maps=False)
```

(continues on next page)

(continued from previous page)

```
>>> LatexExpr(G2.structure_description(latex=True))
C_{6} \times C_{6}
```

This method is mainly intended for small groups or groups with few normal subgroups. Even then there are some surprises:

```
sage: D3 = DihedralGroup(3)                                                    ␣
↪   # needs sage.groups
sage: D3.structure_description()                                               ␣
↪   # needs sage.groups
'S3'
```

```
>>> from sage.all import *
>>> D3 = DihedralGroup(Integer(3))                                             ␣
↪          # needs sage.groups
>>> D3.structure_description()
↪# needs sage.groups
'S3'
```

We use the Sage notation for the degree of dihedral groups:

```
sage: D4 = DihedralGroup(4)                                                    ␣
↪   # needs sage.groups
sage: D4.structure_description()                                               ␣
↪   # needs sage.groups
'D4'
```

```
>>> from sage.all import *
>>> D4 = DihedralGroup(Integer(4))                                             ␣
↪          # needs sage.groups
>>> D4.structure_description()
↪# needs sage.groups
'D4'
```

Works for finitely presented groups (Issue #17573):

```
sage: F.<x, y> = FreeGroup()                                                   ␣
↪   # needs sage.groups
sage: G = F / [x^2*y^-1, x^3*y^2, x*y*x^-1*y^-1]                               ␣
↪   # needs sage.groups
sage: G.structure_description()                                                ␣
↪   # needs sage.groups
'C7'
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('x', 'y',)); (x, y,) = F._first_ngens(2)# needs sage.
↪groups
>>> G = F / [x**Integer(2)*y**-Integer(1), x**Integer(3)*y**Integer(2),␣
↪x*y*x**-Integer(1)*y**-Integer(1)]                                  # needs␣
↪sage.groups
>>> G.structure_description()
```

(continues on next page)

(continued from previous page)

```
↪# needs sage.groups
'C7'
```

And matrix groups (Issue #17573):

```
sage: groups.matrix.GL(4,2).structure_description()                          ␣
↪  # needs sage.libs.gap sage.modules
'A8'
```

```
>>> from sage.all import *
>>> groups.matrix.GL(Integer(4),Integer(2)).structure_description()          ␣
↪                 # needs sage.libs.gap sage.modules
'A8'
```

class sage.groups.finitely_presented.**FinitelyPresentedGroupElement**(*parent*, *x*, *check=True*)

Bases: *FreeGroupElement*

A wrapper of GAP's Finitely Presented Group elements.

The elements are created by passing the Tietze list that determines them.

EXAMPLES:

```
sage: G = FreeGroup('a, b')
sage: H = G / [G([1]), G([2, 2, 2])]
sage: H([1, 2, 1, -1])
a*b
sage: H([1, 2, 1, -2])
a*b*a*b^-1
sage: x = H([1, 2, -1, -2])
sage: x
a*b*a^-1*b^-1
sage: y = H([2, 2, 2, 1, -2, -2, -2])
sage: y
b^3*a*b^-3
sage: x*y
a*b*a^-1*b^2*a*b^-3
sage: x^(-1)
b*a*b^-1*a^-1
```

```
>>> from sage.all import *
>>> G = FreeGroup('a, b')
>>> H = G / [G([Integer(1)]), G([Integer(2), Integer(2), Integer(2)])]
>>> H([Integer(1), Integer(2), Integer(1), -Integer(1)])
a*b
>>> H([Integer(1), Integer(2), Integer(1), -Integer(2)])
a*b*a*b^-1
>>> x = H([Integer(1), Integer(2), -Integer(1), -Integer(2)])
>>> x
a*b*a^-1*b^-1
>>> y = H([Integer(2), Integer(2), Integer(2), Integer(1), -Integer(2), -
↪Integer(2), -Integer(2)])
>>> y
```

(continues on next page)

(continued from previous page)

```
b^3*a*b^-3
>>> x*y
a*b*a^-1*b^2*a*b^-3
>>> x**(-Integer(1))
b*a*b^-1*a^-1
```

Tietze()

Return the Tietze list of the element.

The Tietze list of a word is a list of integers that represent the letters in the word. A positive integer i represents the letter corresponding to the i-th generator of the group. Negative integers represent the inverses of generators.

OUTPUT: tuple of integers

EXAMPLES:

```
sage: G = FreeGroup('a, b')
sage: H = G / (G([1]), G([2, 2, 2]))
sage: H.inject_variables()
Defining a, b
sage: a.Tietze()
(1,)
sage: x = a^2*b^(-3)*a^(-2)
sage: x.Tietze()
(1, 1, -2, -2, -2, -1, -1)
```

```
>>> from sage.all import *
>>> G = FreeGroup('a, b')
>>> H = G / (G([Integer(1)]), G([Integer(2), Integer(2), Integer(2)]))
>>> H.inject_variables()
Defining a, b
>>> a.Tietze()
(1,)
>>> x = a**Integer(2)*b**(-Integer(3))*a**(-Integer(2))
>>> x.Tietze()
(1, 1, -2, -2, -2, -1, -1)
```

class sage.groups.finitely_presented.**GroupMorphismWithGensImages**

Bases: SetMorphism

Class used for morphisms from finitely presented groups to other groups. It just adds the images of the generators at the end of the representation.

EXAMPLES:

```
sage: F = FreeGroup(3)
sage: G = F / [F([1, 2, 3, 1, 2, 3]), F([1, 1, 1])]
sage: H = AlternatingGroup(3)
sage: HS = G.Hom(H)
sage: from sage.groups.finitely_presented import GroupMorphismWithGensImages
sage: GroupMorphismWithGensImages(HS, lambda a: H.one())
Generic morphism:
From: Finitely presented group < x0, x1, x2 | (x0*x1*x2)^2, x0^3 >
```

(continues on next page)

(continued from previous page)

```
To:   Alternating group of order 3!/2 as a permutation group
Defn: x0 |--> ()
      x1 |--> ()
      x2 |--> ()
```

```
>>> from sage.all import *
>>> F = FreeGroup(Integer(3))
>>> G = F / [F([Integer(1), Integer(2), Integer(3), Integer(1), Integer(2),␣
↪Integer(3)]), F([Integer(1), Integer(1), Integer(1)])]
>>> H = AlternatingGroup(Integer(3))
>>> HS = G.Hom(H)
>>> from sage.groups.finitely_presented import GroupMorphismWithGensImages
>>> GroupMorphismWithGensImages(HS, lambda a: H.one())
Generic morphism:
From: Finitely presented group < x0, x1, x2 | (x0*x1*x2)^2, x0^3 >
To:   Alternating group of order 3!/2 as a permutation group
Defn: x0 |--> ()
      x1 |--> ()
      x2 |--> ()
```

class sage.groups.finitely_presented.**RewritingSystem**(G)

Bases: object

A class that wraps GAP's rewriting systems.

A rewriting system is a set of rules that allow to transform one word in the group to an equivalent one.

If the rewriting system is confluent, then the transformed word is a unique reduced form of the element of the group.

Warning

Note that the process of making a rewriting system confluent might not end.

INPUT:

- G – a group

REFERENCES:

- Wikipedia article Knuth-Bendix_completion_algorithm

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G = F / [a*b/a/b]
sage: k = G.rewriting_system()
sage: k
Rewriting system of Finitely presented group < a, b | a*b*a^-1*b^-1 >
with rules:
    a*b*a^-1*b^-1    --->    1

sage: k.reduce(a*b*a*b)
(a*b)^2
```

(continues on next page)

(continued from previous page)

```
sage: k.make_confluent()
sage: k
Rewriting system of Finitely presented group < a, b | a*b*a^-1*b^-1 >
with rules:
    b^-1*a^-1    --->    a^-1*b^-1
    b^-1*a    --->    a*b^-1
    b*a^-1    --->    a^-1*b
    b*a    --->    a*b

sage: k.reduce(a*b*a*b)
a^2*b^2
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('a', 'b',)); (a, b,) = F._first_ngens(2)
>>> G = F / [a*b/a/b]
>>> k = G.rewriting_system()
>>> k
Rewriting system of Finitely presented group < a, b | a*b*a^-1*b^-1 >
with rules:
    a*b*a^-1*b^-1    --->    1

>>> k.reduce(a*b*a*b)
(a*b)^2
>>> k.make_confluent()
>>> k
Rewriting system of Finitely presented group < a, b | a*b*a^-1*b^-1 >
with rules:
    b^-1*a^-1    --->    a^-1*b^-1
    b^-1*a    --->    a*b^-1
    b*a^-1    --->    a^-1*b
    b*a    --->    a*b

>>> k.reduce(a*b*a*b)
a^2*b^2
```

Todo

- Include support for different orderings (currently only shortlex is used).
- Include the GAP package kbmag for more functionalities, including automatic structures and faster compiled functions.

AUTHORS:

- Miguel Angel Marco Buzunariz (2013-12-16)

finitely_presented_group()

The finitely presented group where the rewriting system is defined.

EXAMPLES:

```
sage: F = FreeGroup(3)
sage: G = F / [ [1,2,3], [-1,-2,-3], [1,1], [2,2] ]
sage: k = G.rewriting_system()
sage: k.make_confluent()
sage: k
Rewriting system of Finitely presented group < x0, x1, x2 | x0*x1*x2, x0^-
↪1*x1^-1*x2^-1, x0^2, x1^2 >
with rules:
    x0^-1    --->    x0
    x1^-1    --->    x1
    x2^-1    --->    x2
    x0^2    --->    1
    x0*x1    --->    x2
    x0*x2    --->    x1
    x1*x0    --->    x2
    x1^2    --->    1
    x1*x2    --->    x0
    x2*x0    --->    x1
    x2*x1    --->    x0
    x2^2    --->    1
sage: k.finitely_presented_group()
Finitely presented group < x0, x1, x2 | x0*x1*x2, x0^-1*x1^-1*x2^-1, x0^2, x1^
↪2 >
```

```
>>> from sage.all import *
>>> F = FreeGroup(Integer(3))
>>> G = F / [ [Integer(1),Integer(2),Integer(3)], [-Integer(1),-Integer(2),-
↪Integer(3)], [Integer(1),Integer(1)], [Integer(2),Integer(2)] ]
>>> k = G.rewriting_system()
>>> k.make_confluent()
>>> k
Rewriting system of Finitely presented group < x0, x1, x2 | x0*x1*x2, x0^-
↪1*x1^-1*x2^-1, x0^2, x1^2 >
with rules:
    x0^-1    --->    x0
    x1^-1    --->    x1
    x2^-1    --->    x2
    x0^2    --->    1
    x0*x1    --->    x2
    x0*x2    --->    x1
    x1*x0    --->    x2
    x1^2    --->    1
    x1*x2    --->    x0
    x2*x0    --->    x1
    x2*x1    --->    x0
    x2^2    --->    1
>>> k.finitely_presented_group()
Finitely presented group < x0, x1, x2 | x0*x1*x2, x0^-1*x1^-1*x2^-1, x0^2, x1^
↪2 >
```

free_group()

The free group after which the rewriting system is defined.

EXAMPLES:

```
sage: F = FreeGroup(3)
sage: G = F / [ [1,2,3], [-1,-2,-3] ]
sage: k = G.rewriting_system()
sage: k.free_group()
Free Group on generators {x0, x1, x2}
```

```
>>> from sage.all import *
>>> F = FreeGroup(Integer(3))
>>> G = F / [ [Integer(1),Integer(2),Integer(3)], [-Integer(1),-Integer(2),-
↪Integer(3)] ]
>>> k = G.rewriting_system()
>>> k.free_group()
Free Group on generators {x0, x1, x2}
```

gap()

The gap representation of the rewriting system.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G = F/[a*a,b*b]
sage: k = G.rewriting_system()
sage: k.gap()
Knuth Bendix Rewriting System for Monoid( [ a, A, b, B ] ) with rules
[ [ a*A, <identity ...> ], [ A*a, <identity ...> ],
  [ b*B, <identity ...> ], [ B*b, <identity ...> ],
  [ a^2, <identity ...> ], [ b^2, <identity ...> ] ]
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('a', 'b',)); (a, b,) = F._first_ngens(2)
>>> G = F/[a*a,b*b]
>>> k = G.rewriting_system()
>>> k.gap()
Knuth Bendix Rewriting System for Monoid( [ a, A, b, B ] ) with rules
[ [ a*A, <identity ...> ], [ A*a, <identity ...> ],
  [ b*B, <identity ...> ], [ B*b, <identity ...> ],
  [ a^2, <identity ...> ], [ b^2, <identity ...> ] ]
```

is_confluent()

Return `True` if the system is confluent and `False` otherwise.

EXAMPLES:

```
sage: F = FreeGroup(3)
sage: G = F / [F([1,2,1,2,1,3,-1]),F([2,2,2,1,1,2]),F([1,2,3])]
sage: k = G.rewriting_system()
sage: k.is_confluent()
False
sage: k
Rewriting system of Finitely presented group < x0, x1, x2 | (x0*x1)^
↪2*x0*x2*x0^-1, x1^3*x0^2*x1, x0*x1*x2 >
with rules:
```

(continues on next page)

(continued from previous page)

```
    x0*x1*x2    --->    1
    x1^3*x0^2*x1    --->    1
    (x0*x1)^2*x0*x2*x0^-1    --->    1

sage: k.make_confluent()
sage: k.is_confluent()
True
sage: k
Rewriting system of Finitely presented group < x0, x1, x2 | (x0*x1)^
↪2*x0*x2*x0^-1, x1^3*x0^2*x1, x0*x1*x2 >
with rules:
    x0^-1    --->    x0
    x1^-1    --->    x1
    x0^2    --->    1
    x0*x1    --->    x2^-1
    x0*x2^-1    --->    x1
    x1*x0    --->    x2
    x1^2    --->    1
    x1*x2^-1    --->    x0*x2
    x1*x2    --->    x0
    x2^-1*x0    --->    x0*x2
    x2^-1*x1    --->    x0
    x2^-2    --->    x2
    x2*x0    --->    x1
    x2*x1    --->    x0*x2
    x2^2    --->    x2^-1
```

```
>>> from sage.all import *
>>> F = FreeGroup(Integer(3))
>>> G = F / [F([Integer(1),Integer(2),Integer(1),Integer(2),Integer(1),
↪Integer(3),-Integer(1)]),F([Integer(2),Integer(2),Integer(2),Integer(1),
↪Integer(1),Integer(2)]),F([Integer(1),Integer(2),Integer(3)])]
>>> k = G.rewriting_system()
>>> k.is_confluent()
False
>>> k
Rewriting system of Finitely presented group < x0, x1, x2 | (x0*x1)^
↪2*x0*x2*x0^-1, x1^3*x0^2*x1, x0*x1*x2 >
with rules:
    x0*x1*x2    --->    1
    x1^3*x0^2*x1    --->    1
    (x0*x1)^2*x0*x2*x0^-1    --->    1

>>> k.make_confluent()
>>> k.is_confluent()
True
>>> k
Rewriting system of Finitely presented group < x0, x1, x2 | (x0*x1)^
↪2*x0*x2*x0^-1, x1^3*x0^2*x1, x0*x1*x2 >
with rules:
    x0^-1    --->    x0
    x1^-1    --->    x1
```

(continues on next page)

(continued from previous page)

```
    x0^2    --->    1
    x0*x1    --->    x2^-1
    x0*x2^-1    --->    x1
    x1*x0    --->    x2
    x1^2    --->    1
    x1*x2^-1    --->    x0*x2
    x1*x2    --->    x0
    x2^-1*x0    --->    x0*x2
    x2^-1*x1    --->    x0
    x2^-2    --->    x2
    x2*x0    --->    x1
    x2*x1    --->    x0*x2
    x2^2    --->    x2^-1
```

make_confluent()

Apply the Knuth-Bendix algorithm to try to transform the rewriting system into a confluent one.

Note that this method does not return any object, just changes the rewriting system internally.

> **Warning**
>
> This algorithm is not granted to finish. Although it may be useful in some occasions to run it, interrupt it manually after some time and use then the transformed rewriting system. Even if it is not confluent, it could be used to reduce some words.

ALGORITHM:

Uses GAP's `MakeConfluent`.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G = F / [a^2,b^3,(a*b/a)^3,b*a*b*a]
sage: k = G.rewriting_system()
sage: k
Rewriting system of Finitely presented group < a, b | a^2, b^3, a*b^3*a^-1,
↪(b*a)^2 >
with rules:
    a^2    --->    1
    b^3    --->    1
    (b*a)^2    --->    1
    a*b^3*a^-1    --->    1

sage: k.make_confluent()
sage: k
Rewriting system of Finitely presented group < a, b | a^2, b^3, a*b^3*a^-1,
↪(b*a)^2 >
with rules:
    a^-1    --->    a
    a^2    --->    1
    b^-1*a    --->    a*b
    b^-2    --->    b
```

(continues on next page)

(continued from previous page)

```
    b*a    --->    a*b^-1
    b^2    --->    b^-1
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('a', 'b',)); (a, b,) = F._first_ngens(2)
>>> G = F / [a**Integer(2),b**Integer(3),(a*b/a)**Integer(3),b*a*b*a]
>>> k = G.rewriting_system()
>>> k
Rewriting system of Finitely presented group < a, b | a^2, b^3, a*b^3*a^-1,␣
↪(b*a)^2 >
with rules:
    a^2    --->    1
    b^3    --->    1
    (b*a)^2    --->    1
    a*b^3*a^-1    --->    1

>>> k.make_confluent()
>>> k
Rewriting system of Finitely presented group < a, b | a^2, b^3, a*b^3*a^-1,␣
↪(b*a)^2 >
with rules:
    a^-1    --->    a
    a^2    --->    1
    b^-1*a    --->    a*b
    b^-2    --->    b
    b*a    --->    a*b^-1
    b^2    --->    b^-1
```

reduce(*element*)

Apply the rules in the rewriting system to the element, to obtain a reduced form.

If the rewriting system is confluent, this reduced form is unique for all words representing the same element.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G = F/[a^2, b^3, (a*b/a)^3, b*a*b*a]
sage: k = G.rewriting_system()
sage: k.reduce(b^4)
b
sage: k.reduce(a*b*a)
a*b*a
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('a', 'b',)); (a, b,) = F._first_ngens(2)
>>> G = F/[a**Integer(2), b**Integer(3), (a*b/a)**Integer(3), b*a*b*a]
>>> k = G.rewriting_system()
>>> k.reduce(b**Integer(4))
b
>>> k.reduce(a*b*a)
a*b*a
```

rules()

Return the rules that form the rewriting system.

OUTPUT:

A dictionary containing the rules of the rewriting system. Each key is a word in the free group, and its corresponding value is the word to which it is reduced.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G = F / [a*a*a,b*b*a*a]
sage: k = G.rewriting_system()
sage: k
Rewriting system of Finitely presented group < a, b | a^3, b^2*a^2 >
with rules:
    a^3    --->    1
    b^2*a^2    --->    1

sage: k.rules()
{a^3: 1, b^2*a^2: 1}
sage: k.make_confluent()
sage: sorted(k.rules().items())
[(a^-2, a), (a^-1*b^-1, a*b), (a^-1*b, b^-1), (a^2, a^-1),
 (a*b^-1, b), (b^-1*a^-1, a*b), (b^-1*a, b), (b^-2, a^-1),
 (b*a^-1, b^-1), (b*a, a*b), (b^2, a)]
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('a', 'b',)); (a, b,) = F._first_ngens(2)
>>> G = F / [a*a*a,b*b*a*a]
>>> k = G.rewriting_system()
>>> k
Rewriting system of Finitely presented group < a, b | a^3, b^2*a^2 >
with rules:
    a^3    --->    1
    b^2*a^2    --->    1

>>> k.rules()
{a^3: 1, b^2*a^2: 1}
>>> k.make_confluent()
>>> sorted(k.rules().items())
[(a^-2, a), (a^-1*b^-1, a*b), (a^-1*b, b^-1), (a^2, a^-1),
 (a*b^-1, b), (b^-1*a^-1, a*b), (b^-1*a, b), (b^-2, a^-1),
 (b*a^-1, b^-1), (b*a, a*b), (b^2, a)]
```

CHAPTER

ELEVEN

NAMED FINITELY PRESENTED GROUPS

Construct groups of small order and "named" groups as quotients of free groups. These groups are available through tab completion by typing `groups.presentation.<tab>` or by importing the required methods. Tab completion is made available through Sage's *group catalog*. Some examples are engineered from entries in [TW1980].

Groups available as finite presentations:

- Alternating group, A_n of order $n!/2$ – `groups.presentation.Alternating`
- the n-fruit Cactus group, a standard notation for which is J_n – `groups.presentation.Cactus`
- Cyclic group, C_n of order n – `groups.presentation.Cyclic`
- Dicyclic group, nonabelian groups of order $4n$ with a unique element of order 2 – `groups.presentation.DiCyclic`
- Dihedral group, D_n of order $2n$ – `groups.presentation.Dihedral`
- Finitely generated abelian group, $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \cdots \times \mathbf{Z}_{n_k}$ – `groups.presentation.FGAbelian`
- Finitely generated Heisenberg group – `groups.presentation.Heisenberg`
- Klein four group, $C_2 \times C_2$ – `groups.presentation.KleinFour`
- Quaternion group of order 8 – `groups.presentation.Quaternion`
- Symmetric group, S_n of order $n!$ – `groups.presentation.Symmetric`

AUTHORS:

- Davis Shurbert (2013-06-21): initial version

EXAMPLES:

```
sage: groups.presentation.Cyclic(4)
Finitely presented group < a | a^4 >
```

```
>>> from sage.all import *
>>> groups.presentation.Cyclic(Integer(4))
Finitely presented group < a | a^4 >
```

You can also import the desired functions:

```
sage: from sage.groups.finitely_presented_named import CyclicPresentation
sage: CyclicPresentation(4)
Finitely presented group < a | a^4 >
```

```
>>> from sage.all import *
>>> from sage.groups.finitely_presented_named import CyclicPresentation
>>> CyclicPresentation(Integer(4))
Finitely presented group < a | a^4 >
```

sage.groups.finitely_presented_named.**AlternatingPresentation**(*n*)

Build the Alternating group of order $n!/2$ as a finitely presented group.

INPUT:

- n – the size of the underlying set of arbitrary symbols being acted on by the Alternating group of order $n!/2$

OUTPUT:

Alternating group as a finite presentation, implementation uses GAP to find an isomorphism from a permutation representation to a finitely presented group representation. Due to this fact, the exact output presentation may not be the same for every method call on a constant n.

EXAMPLES:

```
sage: A6 = groups.presentation.Alternating(6)
sage: A6.as_permutation_group().is_isomorphic(AlternatingGroup(6)), A6.order()
(True, 360)
```

```
>>> from sage.all import *
>>> A6 = groups.presentation.Alternating(Integer(6))
>>> A6.as_permutation_group().is_isomorphic(AlternatingGroup(Integer(6))), A6.
↪order()
(True, 360)
```

sage.groups.finitely_presented_named.**BinaryDihedralPresentation**(*n*)

Build a binary dihedral group of order $4n$ as a finitely presented group.

The binary dihedral group BD_n has the following presentation (note that there is a typo in [Sun2010]):

$$BD_n = \langle x, y, z | x^2 = y^2 = z^n = xyz \rangle.$$

INPUT:

- n – the value n

OUTPUT: the binary dihedral group of order $4n$ as finite presentation

EXAMPLES:

```
sage: groups.presentation.BinaryDihedral(9)
Finitely presented group < x, y, z | x^-2*y^2, x^-2*z^9, x^-1*y*z >
```

```
>>> from sage.all import *
>>> groups.presentation.BinaryDihedral(Integer(9))
Finitely presented group < x, y, z | x^-2*y^2, x^-2*z^9, x^-1*y*z >
```

sage.groups.finitely_presented_named.**CactusPresentation**(*n*)

Build the n-fruit cactus group as a finitely presented group.

OUTPUT: Cactus group J_n as a finitely presented group

EXAMPLES:

```
sage: J3 = groups.presentation.Cactus(3); J3                                #␣
↪needs sage.graphs
Finitely presented group < s12, s13, s23 |
 s12^2, s13^2, s23^2, s13*s12*s13^-1*s23^-1, s13*s23*s13^-1*s12^-1 >
```

```
>>> from sage.all import *
>>> J3 = groups.presentation.Cactus(Integer(3)); J3                          ␣
↪     # needs sage.graphs
Finitely presented group < s12, s13, s23 |
 s12^2, s13^2, s23^2, s13*s12*s13^-1*s23^-1, s13*s23*s13^-1*s12^-1 >
```

sage.groups.finitely_presented_named.**CyclicPresentation**(*n*)

Build cyclic group of order n as a finitely presented group.

INPUT:

- n – the order of the cyclic presentation to be returned

OUTPUT: the cyclic group of order n as finite presentation

EXAMPLES:

```
sage: groups.presentation.Cyclic(10)
Finitely presented group < a | a^10 >
sage: n = 8; C = groups.presentation.Cyclic(n)
sage: C.as_permutation_group().is_isomorphic(CyclicPermutationGroup(n))
True
```

```
>>> from sage.all import *
>>> groups.presentation.Cyclic(Integer(10))
Finitely presented group < a | a^10 >
>>> n = Integer(8); C = groups.presentation.Cyclic(n)
>>> C.as_permutation_group().is_isomorphic(CyclicPermutationGroup(n))
True
```

sage.groups.finitely_presented_named.**DiCyclicPresentation**(*n*)

Build the dicyclic group of order $4n$, for $n \geq 2$, as a finitely presented group.

INPUT:

- n – positive integer, 2 or greater, determining the order of the group ($4n$)

OUTPUT: the dicyclic group of order $4n$ is defined by the presentation

$$\langle a, x \mid a^{2n} = 1, x^2 = a^n, x^{-1}ax = a^{-1} \rangle$$

Note

This group is also available as a permutation group via *groups.permutation.DiCyclic*.

EXAMPLES:

```
sage: D = groups.presentation.DiCyclic(9); D
Finitely presented group < a, b | a^18, b^2*a^-9, b^-1*a*b*a >
sage: D.as_permutation_group().is_isomorphic(groups.permutation.DiCyclic(9))
True
```

```
>>> from sage.all import *
>>> D = groups.presentation.DiCyclic(Integer(9)); D
Finitely presented group < a, b | a^18, b^2*a^-9, b^-1*a*b*a >
>>> D.as_permutation_group().is_isomorphic(groups.permutation.
↪DiCyclic(Integer(9)))
True
```

sage.groups.finitely_presented_named.**DihedralPresentation**(*n*)

Build the Dihedral group of order $2n$ as a finitely presented group.

INPUT:

- n – the size of the set that D_n is acting on

OUTPUT: Dihedral group of order $2n$

EXAMPLES:

```
sage: D = groups.presentation.Dihedral(7); D
Finitely presented group < a, b | a^7, b^2, (a*b)^2 >
sage: D.as_permutation_group().is_isomorphic(DihedralGroup(7))
True
```

```
>>> from sage.all import *
>>> D = groups.presentation.Dihedral(Integer(7)); D
Finitely presented group < a, b | a^7, b^2, (a*b)^2 >
>>> D.as_permutation_group().is_isomorphic(DihedralGroup(Integer(7)))
True
```

sage.groups.finitely_presented_named.**FinitelyGeneratedAbelianPresentation**(*int_list*)

Return canonical presentation of finitely generated abelian group.

INPUT:

- int_list – list of integers defining the group to be returned, the defining list is reduced to the invariants of the input list before generating the corresponding group

OUTPUT:

Finitely generated abelian group, $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \cdots \times \mathbf{Z}_{n_k}$ as a finite presentation, where n_i forms the invariants of the input list.

EXAMPLES:

```
sage: groups.presentation.FGAbelian([2,2])
Finitely presented group < a, b | a^2, b^2, a^-1*b^-1*a*b >
sage: groups.presentation.FGAbelian([2,3])
Finitely presented group < a | a^6 >
sage: groups.presentation.FGAbelian([2,4])
Finitely presented group < a, b | a^2, b^4, a^-1*b^-1*a*b >
```

```
>>> from sage.all import *
>>> groups.presentation.FGAbelian([Integer(2),Integer(2)])
Finitely presented group < a, b | a^2, b^2, a^-1*b^-1*a*b >
>>> groups.presentation.FGAbelian([Integer(2),Integer(3)])
Finitely presented group < a | a^6 >
>>> groups.presentation.FGAbelian([Integer(2),Integer(4)])
Finitely presented group < a, b | a^2, b^4, a^-1*b^-1*a*b >
```

You can create free abelian groups:

```
sage: groups.presentation.FGAbelian([0])
Finitely presented group < a |  >
sage: groups.presentation.FGAbelian([0,0])
Finitely presented group < a, b | a^-1*b^-1*a*b >
sage: groups.presentation.FGAbelian([0,0,0])
Finitely presented group < a, b, c | a^-1*b^-1*a*b, a^-1*c^-1*a*c, b^-1*c^-1*b*c >
```

```
>>> from sage.all import *
>>> groups.presentation.FGAbelian([Integer(0)])
Finitely presented group < a |  >
>>> groups.presentation.FGAbelian([Integer(0),Integer(0)])
Finitely presented group < a, b | a^-1*b^-1*a*b >
>>> groups.presentation.FGAbelian([Integer(0),Integer(0),Integer(0)])
Finitely presented group < a, b, c | a^-1*b^-1*a*b, a^-1*c^-1*a*c, b^-1*c^-1*b*c >
```

And various infinite abelian groups:

```
sage: groups.presentation.FGAbelian([0,2])
Finitely presented group < a, b | a^2, a^-1*b^-1*a*b >
sage: groups.presentation.FGAbelian([0,2,2])
Finitely presented group < a, b, c | a^2, b^2, a^-1*b^-1*a*b, a^-1*c^-1*a*c, b^-
↪1*c^-1*b*c >
```

```
>>> from sage.all import *
>>> groups.presentation.FGAbelian([Integer(0),Integer(2)])
Finitely presented group < a, b | a^2, a^-1*b^-1*a*b >
>>> groups.presentation.FGAbelian([Integer(0),Integer(2),Integer(2)])
Finitely presented group < a, b, c | a^2, b^2, a^-1*b^-1*a*b, a^-1*c^-1*a*c, b^-
↪1*c^-1*b*c >
```

Outputs are reduced to minimal generators and relations:

```
sage: groups.presentation.FGAbelian([3,5,2,7,3])
Finitely presented group < a, b | a^3, b^210, a^-1*b^-1*a*b >
sage: groups.presentation.FGAbelian([3,210])
Finitely presented group < a, b | a^3, b^210, a^-1*b^-1*a*b >
```

```
>>> from sage.all import *
>>> groups.presentation.FGAbelian([Integer(3),Integer(5),Integer(2),Integer(7),
↪Integer(3)])
Finitely presented group < a, b | a^3, b^210, a^-1*b^-1*a*b >
>>> groups.presentation.FGAbelian([Integer(3),Integer(210)])
Finitely presented group < a, b | a^3, b^210, a^-1*b^-1*a*b >
```

The trivial group is an acceptable output:

```
sage: groups.presentation.FGAbelian([])
Finitely presented group <  |  >
sage: groups.presentation.FGAbelian([1])
Finitely presented group <  |  >
sage: groups.presentation.FGAbelian([1,1,1,1,1,1,1,1,1,1])
Finitely presented group <  |  >
```

```
>>> from sage.all import *
>>> groups.presentation.FGAbelian([])
Finitely presented group <  |  >
>>> groups.presentation.FGAbelian([Integer(1)])
Finitely presented group <  |  >
>>> groups.presentation.FGAbelian([Integer(1),Integer(1),Integer(1),Integer(1),
↪Integer(1),Integer(1),Integer(1),Integer(1),Integer(1),Integer(1)])
Finitely presented group <  |  >
```

Input list must consist of positive integers:

```
sage: groups.presentation.FGAbelian([2,6,3,9,-4])
Traceback (most recent call last):
...
ValueError: input list must contain nonnegative entries
sage: groups.presentation.FGAbelian([2,'a',4])
Traceback (most recent call last):
...
TypeError: unable to convert 'a' to an integer
```

```
>>> from sage.all import *
>>> groups.presentation.FGAbelian([Integer(2),Integer(6),Integer(3),Integer(9),-
↪Integer(4)])
Traceback (most recent call last):
...
ValueError: input list must contain nonnegative entries
>>> groups.presentation.FGAbelian([Integer(2),'a',Integer(4)])
Traceback (most recent call last):
...
TypeError: unable to convert 'a' to an integer
```

sage.groups.finitely_presented_named.**FinitelyGeneratedHeisenbergPresentation**(*n=1*, *p=0*)

Return a finite presentation of the Heisenberg group.

The Heisenberg group is the group of $(n+2)\times(n+2)$ matrices over a ring R with diagonal elements equal to 1, first row and last column possibly nonzero, and all the other entries equal to zero.

INPUT:

- n – the degree of the Heisenberg group
- p – (optional) a prime number, where we construct the Heisenberg group over the finite field $\mathbf{Z}/p\mathbf{Z}$

OUTPUT: finitely generated Heisenberg group over the finite field of order p or over the integers

See also

HeisenbergGroup

EXAMPLES:

```
sage: H = groups.presentation.Heisenberg(); H
Finitely presented group < x1, y1, z |
 x1*y1*x1^-1*y1^-1*z^-1, z*x1*z^-1*x1^-1, z*y1*z^-1*y1^-1 >
sage: H.order()
+Infinity
sage: r1, r2, r3 = H.relations()
sage: A = matrix([[1, 1, 0], [0, 1, 0], [0, 0, 1]])
sage: B = matrix([[1, 0, 0], [0, 1, 1], [0, 0, 1]])
sage: C = matrix([[1, 0, 1], [0, 1, 0], [0, 0, 1]])
sage: r1(A, B, C)
[1 0 0]
[0 1 0]
[0 0 1]
sage: r2(A, B, C)
[1 0 0]
[0 1 0]
[0 0 1]
sage: r3(A, B, C)
[1 0 0]
[0 1 0]
[0 0 1]
sage: p = 3
sage: Hp = groups.presentation.Heisenberg(p=3)
sage: Hp.order() == p**3
True
sage: Hnp = groups.presentation.Heisenberg(n=2, p=3)
sage: len(Hnp.relations())
13
```

```
>>> from sage.all import *
>>> H = groups.presentation.Heisenberg(); H
Finitely presented group < x1, y1, z |
 x1*y1*x1^-1*y1^-1*z^-1, z*x1*z^-1*x1^-1, z*y1*z^-1*y1^-1 >
>>> H.order()
+Infinity
>>> r1, r2, r3 = H.relations()
>>> A = matrix([[Integer(1), Integer(1), Integer(0)], [Integer(0), Integer(1),␣
↪Integer(0)], [Integer(0), Integer(0), Integer(1)]])
>>> B = matrix([[Integer(1), Integer(0), Integer(0)], [Integer(0), Integer(1),␣
↪Integer(1)], [Integer(0), Integer(0), Integer(1)]])
>>> C = matrix([[Integer(1), Integer(0), Integer(1)], [Integer(0), Integer(1),␣
↪Integer(0)], [Integer(0), Integer(0), Integer(1)]])
>>> r1(A, B, C)
[1 0 0]
[0 1 0]
[0 0 1]
```

(continues on next page)

(continued from previous page)

```
>>> r2(A, B, C)
[1 0 0]
[0 1 0]
[0 0 1]
>>> r3(A, B, C)
[1 0 0]
[0 1 0]
[0 0 1]
>>> p = Integer(3)
>>> Hp = groups.presentation.Heisenberg(p=Integer(3))
>>> Hp.order() == p**Integer(3)
True
>>> Hnp = groups.presentation.Heisenberg(n=Integer(2), p=Integer(3))
>>> len(Hnp.relations())
13
```

REFERENCES:

- Wikipedia article Heisenberg_group

sage.groups.finitely_presented_named.**KleinFourPresentation**()

Build the Klein group of order 4 as a finitely presented group.

OUTPUT: Klein four group ($C_2 \times C_2$) as a finitely presented group

EXAMPLES:

```
sage: K = groups.presentation.KleinFour(); K
Finitely presented group < a, b | a^2, b^2, a^-1*b^-1*a*b >
```

```
>>> from sage.all import *
>>> K = groups.presentation.KleinFour(); K
Finitely presented group < a, b | a^2, b^2, a^-1*b^-1*a*b >
```

sage.groups.finitely_presented_named.**QuaternionPresentation**()

Build the Quaternion group of order 8 as a finitely presented group.

OUTPUT: Quaternion group as a finite presentation

EXAMPLES:

```
sage: Q = groups.presentation.Quaternion(); Q
Finitely presented group < a, b | a^4, b^2*a^-2, a*b*a*b^-1 >
sage: Q.as_permutation_group().is_isomorphic(QuaternionGroup())
True
```

```
>>> from sage.all import *
>>> Q = groups.presentation.Quaternion(); Q
Finitely presented group < a, b | a^4, b^2*a^-2, a*b*a*b^-1 >
>>> Q.as_permutation_group().is_isomorphic(QuaternionGroup())
True
```

sage.groups.finitely_presented_named.**SymmetricPresentation**(n)

Build the Symmetric group of order $n!$ as a finitely presented group.

INPUT:

- n – the size of the underlying set of arbitrary symbols being acted on by the Symmetric group of order $n!$

OUTPUT:

Symmetric group as a finite presentation, implementation uses GAP to find an isomorphism from a permutation representation to a finitely presented group representation. Due to this fact, the exact output presentation may not be the same for every method call on a constant n.

EXAMPLES:

```
sage: S4 = groups.presentation.Symmetric(4)
sage: S4.as_permutation_group().is_isomorphic(SymmetricGroup(4))
True
```

```
>>> from sage.all import *
>>> S4 = groups.presentation.Symmetric(Integer(4))
>>> S4.as_permutation_group().is_isomorphic(SymmetricGroup(Integer(4)))
True
```

CHAPTER

TWELVE

BRAID GROUPS

Braid groups are implemented as a particular case of finitely presented groups, but with a lot of specific methods for braids.

A braid group can be created by giving the number of strands, and the name of the generators:

```
sage: BraidGroup(3)
Braid group on 3 strands
sage: BraidGroup(3,'a')
Braid group on 3 strands
sage: BraidGroup(3,'a').gens()
(a0, a1)
sage: BraidGroup(3,'a,b').gens()
(a, b)
```

```
>>> from sage.all import *
>>> BraidGroup(Integer(3))
Braid group on 3 strands
>>> BraidGroup(Integer(3),'a')
Braid group on 3 strands
>>> BraidGroup(Integer(3),'a').gens()
(a0, a1)
>>> BraidGroup(Integer(3),'a,b').gens()
(a, b)
```

The elements can be created by operating with the generators, or by passing a list with the indices of the letters to the group:

```
sage: B.<s0,s1,s2> = BraidGroup(4)
sage: s0*s1*s0
s0*s1*s0
sage: B([1,2,1])
s0*s1*s0
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4), names=('s0', 's1', 's2',)); (s0, s1, s2,) = B._first_
↪ngens(3)
>>> s0*s1*s0
s0*s1*s0
>>> B([Integer(1),Integer(2),Integer(1)])
s0*s1*s0
```

The mapping class action of the braid group over the free group is also implemented, see *MappingClassGroupAction* for an explanation. This action is left multiplication of a free group element by a braid:

```
sage: B.<b0,b1,b2> = BraidGroup()
sage: F.<f0,f1,f2,f3> = FreeGroup()
sage: B.strands() == F.rank()    # necessary for the action to be defined
True
sage: f1 * b1
f1*f2*f1^-1
sage: f0 * b1
f0
sage: f1 * b1
f1*f2*f1^-1
sage: f1^-1 * b1
f1*f2^-1*f1^-1
```

```
>>> from sage.all import *
>>> B = BraidGroup(names=('b0', 'b1', 'b2',)); (b0, b1, b2,) = B._first_ngens(3)
>>> F = FreeGroup(names=('f0', 'f1', 'f2', 'f3',)); (f0, f1, f2, f3,) = F._first_
↪ngens(4)
>>> B.strands() == F.rank()    # necessary for the action to be defined
True
>>> f1 * b1
f1*f2*f1^-1
>>> f0 * b1
f0
>>> f1 * b1
f1*f2*f1^-1
>>> f1**-Integer(1) * b1
f1*f2^-1*f1^-1
```

AUTHORS:

- Miguel Angel Marco Buzunariz
- Volker Braun
- Søren Fuglede Jørgensen
- Robert Lipshitz
- Thierry Monteil: add a `__hash__` method consistent with the word problem to ensure correct Cayley graph computations.
- Sebastian Oehms (July and Nov 2018): add other versions for burau_matrix (unitary + simple, see Issue #25760 and Issue #26657)
- Moritz Firsching (Sept 2021): Colored Jones polynomial
- Sebastian Oehms (May 2022): add `links_gould_polynomial()`

class `sage.groups.braid.`**`Braid`**(*parent*, *x*, *check=True*)

Bases: *FiniteTypeArtinGroupElement*

An element of a braid group.

It is a particular case of element of a finitely presented group.

EXAMPLES:

```
sage: B.<s0,s1,s2> = BraidGroup(4)
sage: B
Braid group on 4 strands
sage: s0*s1/s2/s1
s0*s1*s2^-1*s1^-1
sage: B((1, 2, -3, -2))
s0*s1*s2^-1*s1^-1
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4), names=('s0', 's1', 's2',)); (s0, s1, s2,) = B._
↪first_ngens(3)
>>> B
Braid group on 4 strands
>>> s0*s1/s2/s1
s0*s1*s2^-1*s1^-1
>>> B((Integer(1), Integer(2), -Integer(3), -Integer(2)))
s0*s1*s2^-1*s1^-1
```

LKB_matrix(*variables='x,y'*)

Return the Lawrence-Krammer-Bigelow representation matrix.

The matrix is expressed in the basis $\{e_{i,j} \mid 1 \leq i < j \leq n\}$, where the indices are ordered lexicographically. It is a matrix whose entries are in the ring of Laurent polynomials on the given variables. By default, the variables are `'x'` and `'y'`.

INPUT:

- `variables` – string (default: `'x,y'`); a string containing the names of the variables, separated by a comma

OUTPUT: the matrix corresponding to the Lawrence-Krammer-Bigelow representation of the braid

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: b = B([1, 2, 1])
sage: b.LKB_matrix()
[             0 -x^4*y + x^3*y         -x^4*y]
[             0         -x^3*y              0]
[        -x^2*y  x^3*y - x^2*y              0]
sage: c = B([2, 1, 2])
sage: c.LKB_matrix()
[             0 -x^4*y + x^3*y         -x^4*y]
[             0         -x^3*y              0]
[        -x^2*y  x^3*y - x^2*y              0]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> b = B([Integer(1), Integer(2), Integer(1)])
>>> b.LKB_matrix()
[             0 -x^4*y + x^3*y         -x^4*y]
[             0         -x^3*y              0]
[        -x^2*y  x^3*y - x^2*y              0]
>>> c = B([Integer(2), Integer(1), Integer(2)])
>>> c.LKB_matrix()
```

(continues on next page)

(continued from previous page)

```
[            0 -x^4*y + x^3*y          -x^4*y]
[            0         -x^3*y               0]
[       -x^2*y  x^3*y - x^2*y               0]
```

REFERENCES:

- [Big2003]

TL_matrix(*drain_size*, *variab=None*, *sparse=True*)

Return the matrix representation of the Temperley–Lieb–Jones representation of the braid in a certain basis.

The basis is given by non-intersecting pairings of $(n+d)$ points, where n is the number of strands, d is given by `drain_size`, and the pairings satisfy certain rules. See `TL_basis_with_drain()` for details.

We use the convention that the eigenvalues of the standard generators are 1 and $-A^4$, where A is a variable of a Laurent polynomial ring.

When $d = n - 2$ and the variables are picked appropriately, the resulting representation is equivalent to the reduced Burau representation.

INPUT:

- `drain_size` – integer between 0 and the number of strands (both inclusive)
- `variab` – variable (default: `None`); the variable in the entries of the matrices; if `None`, then use a default variable in $\mathbf{Z}[A, A^{-1}]$
- `sparse` – boolean (default: `True`); whether or not the result should be given as a sparse matrix

OUTPUT: the matrix of the TL representation of the braid

The parameter `sparse` can be set to `False` if it is expected that the resulting matrix will not be sparse. We currently make no attempt at guessing this.

EXAMPLES:

Let us calculate a few examples for B_4 with $d = 0$:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, -3])
sage: b.TL_matrix(0)
[1 - A^4   -A^-2]
[   -A^6       0]
sage: R.<x> = LaurentPolynomialRing(GF(2))
sage: b.TL_matrix(0, variab=x)
[1 + x^4    x^-2]
[    x^6       0]
sage: b = B([])
sage: b.TL_matrix(0)
[1 0]
[0 1]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1), Integer(2), -Integer(3)])
>>> b.TL_matrix(Integer(0))
[1 - A^4   -A^-2]
[   -A^6       0]
```

(continues on next page)

(continued from previous page)

```
>>> R = LaurentPolynomialRing(GF(Integer(2)), names=('x',)); (x,) = R._first_
↪ngens(1)
>>> b.TL_matrix(Integer(0), variab=x)
[1 + x^4    x^-2]
[   x^6        0]
>>> b = B([])
>>> b.TL_matrix(Integer(0))
[1 0]
[0 1]
```

Test of one of the relations in B_8:

```
sage: B = BraidGroup(8)
sage: d = 0
sage: B([4,5,4]).TL_matrix(d) == B([5,4,5]).TL_matrix(d)
True
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(8))
>>> d = Integer(0)
>>> B([Integer(4),Integer(5),Integer(4)]).TL_matrix(d) == B([Integer(5),
↪Integer(4),Integer(5)]).TL_matrix(d)
True
```

An element of the kernel of the Burau representation, following [Big1999]:

```
sage: B = BraidGroup(6)
sage: psi1 = B([4, -5, -2, 1])
sage: psi2 = B([-4, 5, 5, 2, -1, -1])
sage: w1 = psi1^(-1) * B([3]) * psi1
sage: w2 = psi2^(-1) * B([3]) * psi2
sage: (w1 * w2 * w1^(-1) * w2^(-1)).TL_matrix(4)
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(6))
>>> psi1 = B([Integer(4), -Integer(5), -Integer(2), Integer(1)])
>>> psi2 = B([-Integer(4), Integer(5), Integer(5), Integer(2), -Integer(1), -
↪Integer(1)])
>>> w1 = psi1**(-Integer(1)) * B([Integer(3)]) * psi1
>>> w2 = psi2**(-Integer(1)) * B([Integer(3)]) * psi2
>>> (w1 * w2 * w1**(-Integer(1)) * w2**(-Integer(1))).TL_matrix(Integer(4))
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

REFERENCES:

- [Big1999]
- [Jon2005]

alexander_polynomial(*var='t'*, *normalized=True*)

Return the Alexander polynomial of the closure of the braid.

INPUT:

- `var` – string (default: `'t'`); the name of the variable in the entries of the matrix
- `normalized` – boolean (default: `True`); whether to return the normalized Alexander polynomial

OUTPUT: the Alexander polynomial of the braid closure of the braid

This is computed using the reduced Burau representation. The unnormalized Alexander polynomial is a Laurent polynomial, which is only well-defined up to multiplication by plus or minus times a power of t.

We normalize the polynomial by dividing by the largest power of t and then if the resulting constant coefficient is negative, we multiply by -1.

EXAMPLES:

We first construct the trefoil:

```
sage: B = BraidGroup(3)
sage: b = B([1,2,1,2])
sage: b.alexander_polynomial(normalized=False)
1 - t + t^2
sage: b.alexander_polynomial()
t^-2 - t^-1 + 1
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> b = B([Integer(1),Integer(2),Integer(1),Integer(2)])
>>> b.alexander_polynomial(normalized=False)
1 - t + t^2
>>> b.alexander_polynomial()
t^-2 - t^-1 + 1
```

Next we construct the figure 8 knot:

```
sage: b = B([-1,2,-1,2])
sage: b.alexander_polynomial(normalized=False)
-t^-2 + 3*t^-1 - 1
sage: b.alexander_polynomial()
t^-2 - 3*t^-1 + 1
```

```
>>> from sage.all import *
>>> b = B([-Integer(1),Integer(2),-Integer(1),Integer(2)])
>>> b.alexander_polynomial(normalized=False)
-t^-2 + 3*t^-1 - 1
>>> b.alexander_polynomial()
t^-2 - 3*t^-1 + 1
```

Our last example is the Kinoshita-Terasaka knot:

```
sage: B = BraidGroup(4)
sage: b = B([1,1,1,3,3,2,-3,-1,-1,2,-1,-3,-2])
sage: b.alexander_polynomial(normalized=False)
-t^-1
sage: b.alexander_polynomial()
1
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1),Integer(1),Integer(1),Integer(3),Integer(3),Integer(2),-
↪Integer(3),-Integer(1),-Integer(1),Integer(2),-Integer(1),-Integer(3),-
↪Integer(2)])
>>> b.alexander_polynomial(normalized=False)
-t^-1
>>> b.alexander_polynomial()
1
```

REFERENCES:

- Wikipedia article Alexander_polynomial

annular_khovanov_complex(*qagrad=None*, *ring=None*)

Return the annular Khovanov complex of the closure of a braid, as defined in [BG2013].

INPUT:

- `qagrad` – tuple of quantum and annular grading for which to compute the chain complex; if not specified all gradings are computed
- `ring` – (default: `ZZ`) the coefficient ring

OUTPUT:

The annular Khovanov complex of the braid, given as a dictionary whose keys are tuples of quantum and annular grading. If `qagrad` is specified only return the chain complex of that grading.

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: b = B([1,-2,1,-2])
sage: C = b.annular_khovanov_complex()
sage: C
{(-5, -1): Chain complex with at most 1 nonzero terms over Integer Ring,
 (-3, -3): Chain complex with at most 1 nonzero terms over Integer Ring,
 (-3, -1): Chain complex with at most 2 nonzero terms over Integer Ring,
 (-3, 1): Chain complex with at most 1 nonzero terms over Integer Ring,
 (-1, -1): Chain complex with at most 5 nonzero terms over Integer Ring,
 (-1, 1): Chain complex with at most 2 nonzero terms over Integer Ring,
 (1, -1): Chain complex with at most 2 nonzero terms over Integer Ring,
 (1, 1): Chain complex with at most 5 nonzero terms over Integer Ring,
 (3, -1): Chain complex with at most 1 nonzero terms over Integer Ring,
 (3, 1): Chain complex with at most 2 nonzero terms over Integer Ring,
 (3, 3): Chain complex with at most 1 nonzero terms over Integer Ring,
 (5, 1): Chain complex with at most 1 nonzero terms over Integer Ring}
sage: C[1,-1].homology()
{1: Z x Z, 2: 0}
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> b = B([Integer(1),-Integer(2),Integer(1),-Integer(2)])
>>> C = b.annular_khovanov_complex()
>>> C
{(-5, -1): Chain complex with at most 1 nonzero terms over Integer Ring,
 (-3, -3): Chain complex with at most 1 nonzero terms over Integer Ring,
 (-3, -1): Chain complex with at most 2 nonzero terms over Integer Ring,
 (-3, 1): Chain complex with at most 1 nonzero terms over Integer Ring,
 (-1, -1): Chain complex with at most 5 nonzero terms over Integer Ring,
 (-1, 1): Chain complex with at most 2 nonzero terms over Integer Ring,
 (1, -1): Chain complex with at most 2 nonzero terms over Integer Ring,
 (1, 1): Chain complex with at most 5 nonzero terms over Integer Ring,
 (3, -1): Chain complex with at most 1 nonzero terms over Integer Ring,
 (3, 1): Chain complex with at most 2 nonzero terms over Integer Ring,
 (3, 3): Chain complex with at most 1 nonzero terms over Integer Ring,
 (5, 1): Chain complex with at most 1 nonzero terms over Integer Ring}
>>> C[Integer(1),-Integer(1)].homology()
{1: Z x Z, 2: 0}
```

annular_khovanov_homology(*qagrad=None*, *ring=Integer Ring*)

Return the annular Khovanov homology of a closure of a braid.

INPUT:

- `qagrad` – (optional) tuple of quantum and annular grading for which to compute the homology
- `ring` – (default: `ZZ`) the coefficient ring

OUTPUT:

If `qagrad` is `None`, return a dictionary of homologies in all gradings indexed by grading. If `qagrad` is specified, return the homology of that grading.

Note

This is a simple wrapper around *annular_khovanov_complex()* to compute homology from it.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([1,3,-2])
sage: b.annular_khovanov_homology()
{(-3, -4): {0: Z},
 (-3, -2): {-1: Z},
 (-1, -2): {-1: 0, 0: Z x Z x Z, 1: 0},
 (-1, 0): {-1: Z x Z},
 (1, -2): {1: Z x Z},
 (1, 0): {-1: 0, 0: Z x Z x Z x Z, 1: 0, 2: 0},
 (1, 2): {-1: Z},
 (3, 0): {1: Z x Z x Z, 2: 0},
 (3, 2): {-1: 0, 0: Z x Z x Z, 1: 0},
 (5, 0): {2: Z},
 (5, 2): {1: Z x Z},
```

(continues on next page)

(continued from previous page)

```
 (5, 4): {0: Z}}

sage: B = BraidGroup(2)
sage: b = B([1,1,1])
sage: b.annular_khovanov_homology((7,0))
{2: 0, 3: C2}
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1),Integer(3),-Integer(2)])
>>> b.annular_khovanov_homology()
{(-3, -4): {0: Z},
 (-3, -2): {-1: Z},
 (-1, -2): {-1: 0, 0: Z x Z x Z, 1: 0},
 (-1, 0): {-1: Z x Z},
 (1, -2): {1: Z x Z},
 (1, 0): {-1: 0, 0: Z x Z x Z x Z, 1: 0, 2: 0},
 (1, 2): {-1: Z},
 (3, 0): {1: Z x Z x Z, 2: 0},
 (3, 2): {-1: 0, 0: Z x Z x Z, 1: 0},
 (5, 0): {2: Z},
 (5, 2): {1: Z x Z},
 (5, 4): {0: Z}}

>>> B = BraidGroup(Integer(2))
>>> b = B([Integer(1),Integer(1),Integer(1)])
>>> b.annular_khovanov_homology((Integer(7),Integer(0)))
{2: 0, 3: C2}
```

burau_matrix(*var='t'*, *reduced=False*)

Return the Burau matrix of the braid.

INPUT:

- `var` – string (default: `'t'`); the name of the variable in the entries of the matrix
- `reduced` – boolean (default: `False`); whether to return the reduced or unreduced Burau representation, can be one of the following:
  - `True` or `'increasing'` – returns the reduced form using the basis given by $e_1 - e_i$ for $2 \leq i \leq n$
  - `'unitary'` – the unitary form according to Squier [Squ1984]
  - `'simple'` – returns the reduced form using the basis given by simple roots $e_i - e_{i+1}$, which yields the matrices given on the Wikipedia page

OUTPUT:

The Burau matrix of the braid. It is a matrix whose entries are Laurent polynomials in the variable `var`. If `reduced` is `True`, return the matrix for the reduced Burau representation instead in the format specified. If `reduced` is `'unitary'`, a triple `M, Madj, H` is returned, where `M` is the Burau matrix in the unitary form, `Madj` the adjoined to `M` and `H` the hermitian form.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: B.inject_variables()
Defining s0, s1, s2
sage: b = s0 * s1 / s2 / s1
sage: b.burau_matrix()
[       1 - t              0      t - t^2          t^2]
[           1              0            0            0]
[           0              0            1            0]
[           0          t^-2 -t^-2 + t^-1    -t^-1 + 1]
sage: s2.burau_matrix('x')
[    1     0     0     0]
[    0     1     0     0]
[    0     0 1 - x     x]
[    0     0     1     0]
sage: s0.burau_matrix(reduced=True)
[-t  0  0]
[-t  1  0]
[-t  0  1]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> B.inject_variables()
Defining s0, s1, s2
>>> b = s0 * s1 / s2 / s1
>>> b.burau_matrix()
[       1 - t              0      t - t^2          t^2]
[           1              0            0            0]
[           0              0            1            0]
[           0          t^-2 -t^-2 + t^-1    -t^-1 + 1]
>>> s2.burau_matrix('x')
[    1     0     0     0]
[    0     1     0     0]
[    0     0 1 - x     x]
[    0     0     1     0]
>>> s0.burau_matrix(reduced=True)
[-t  0  0]
[-t  1  0]
[-t  0  1]
```

Using the different reduced forms:

```
sage: b.burau_matrix(reduced='simple')
[    1 - t -t^-1 + 1        -1]
[        1 -t^-1 + 1        -1]
[        1     -t^-1         0]

sage: M, Madj, H = b.burau_matrix(reduced='unitary')
sage: M
[-t^-2 + 1          t       t^2]
[ t^-1 - t    1 - t^2      -t^3]
[    -t^-2      -t^-1         0]
sage: Madj
[  1 - t^2 -t^-1 + t      -t^2]
```

(continues on next page)

(continued from previous page)

```
[      t^-1 -t^-2 + 1          -t]
[      t^-2      -t^-3          0]
sage: H
[t^-1 + t         -1          0]
[       -1 t^-1 + t         -1]
[        0         -1 t^-1 + t]
sage: M * H * Madj == H
True
```

```
>>> from sage.all import *
>>> b.burau_matrix(reduced='simple')
[    1 - t -t^-1 + 1          -1]
[        1 -t^-1 + 1          -1]
[        1      -t^-1          0]

>>> M, Madj, H = b.burau_matrix(reduced='unitary')
>>> M
[-t^-2 + 1          t        t^2]
[ t^-1 - t    1 - t^2       -t^3]
[    -t^-2      -t^-1          0]
>>> Madj
[  1 - t^2 -t^-1 + t       -t^2]
[      t^-1 -t^-2 + 1          -t]
[      t^-2      -t^-3          0]
>>> H
[t^-1 + t         -1          0]
[       -1 t^-1 + t         -1]
[        0         -1 t^-1 + t]
>>> M * H * Madj == H
True
```

The adjoined matrix (`Madj` in the above example) matches the output of *sage.groups.artin.ArtinGroupElement.burau_matrix()*:

```
sage: from sage.groups.artin import ArtinGroupElement
sage: Madj == ArtinGroupElement.burau_matrix(b)
True

sage: a = s0^2 * s1 * s0 * s2 *s1 * ~s0 * s1^3 * s0 * s2 * s1^-2 * s0
sage: a.burau_matrix(reduced='unitary')[1] == ArtinGroupElement.burau_
↪matrix(a)
True
```

```
>>> from sage.all import *
>>> from sage.groups.artin import ArtinGroupElement
>>> Madj == ArtinGroupElement.burau_matrix(b)
True

>>> a = s0**Integer(2) * s1 * s0 * s2 *s1 * ~s0 * s1**Integer(3) * s0 * s2 *␣
↪s1**-Integer(2) * s0
>>> a.burau_matrix(reduced='unitary')[Integer(1)] == ArtinGroupElement.burau_
↪matrix(a)
```

(continues on next page)

(continued from previous page)

```
True
```

We verify Bigelow's example that in B_5 the Burau representation is not faithful:

```
sage: B.<s1,s2,s3,s4> = BraidGroup(5)
sage: psi1 = ~s3 * s2 * s1^2 * s2 * s4^3 * s3 * s2
sage: psi2 = ~s4 * s3 * s2 * s1^-2 * s2 * s1^2 * s2^2 * s1 * s4^5
sage: alpha = ~psi1 * s4 * psi1
sage: beta = ~psi2 * s4 * s3 * s2 * s1^2 * s2 * s3 * s4 * psi2
sage: elm = alpha * beta * ~alpha * ~beta
sage: elm.burau_matrix()
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
sage: elm.burau_matrix(reduced=True)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: elm.is_one()
False
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(5), names=('s1', 's2', 's3', 's4',)); (s1, s2, s3,␣
↪s4,) = B._first_ngens(4)
>>> psi1 = ~s3 * s2 * s1**Integer(2) * s2 * s4**Integer(3) * s3 * s2
>>> psi2 = ~s4 * s3 * s2 * s1**-Integer(2) * s2 * s1**Integer(2) *␣
↪s2**Integer(2) * s1 * s4**Integer(5)
>>> alpha = ~psi1 * s4 * psi1
>>> beta = ~psi2 * s4 * s3 * s2 * s1**Integer(2) * s2 * s3 * s4 * psi2
>>> elm = alpha * beta * ~alpha * ~beta
>>> elm.burau_matrix()
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
>>> elm.burau_matrix(reduced=True)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
>>> elm.is_one()
False
```

REFERENCES:

- Wikipedia article Burau_representation
- [Squ1984]

centralizer()

Return a list of generators of the centralizer of the braid.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([2, 1, 3, 2])
sage: b.centralizer()
[s1*s0*s2*s1, s0*s2]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(2), Integer(1), Integer(3), Integer(2)])
>>> b.centralizer()
[s1*s0*s2*s1, s0*s2]
```

colored_jones_polynomial(*N*, *variab=None*, *try_inverse=True*)

Return the colored Jones polynomial of the trace closure of the braid.

INPUT:

- `N` – integer; the number of colors
- `variab` – (default: q) the variable in the resulting Laurent polynomial
- `try_inverse` – boolean (default: `True`); if `True`, attempt a faster calculation by using the inverse of the braid

ALGORITHM:

The algorithm used is described in [HL2018]. We follow their notation, but work in a suitable free algebra over a Laurent polynomial ring in one variable to simplify bookkeeping.

EXAMPLES:

```
sage: trefoil = BraidGroup(2)([1,1,1])
sage: trefoil.colored_jones_polynomial(2)
q + q^3 - q^4
sage: trefoil.colored_jones_polynomial(4)
q^3 + q^7 - q^10 + q^11 - q^13 - q^14 + q^15 - q^17
 + q^19 + q^20 - q^21
sage: trefoil.inverse().colored_jones_polynomial(4)
-q^-21 + q^-20 + q^-19 - q^-17 + q^-15 - q^-14 - q^-13
 + q^-11 - q^-10 + q^-7 + q^-3

sage: figure_eight = BraidGroup(3)([-1, 2, -1, 2])
sage: figure_eight.colored_jones_polynomial(2)
q^-2 - q^-1 + 1 - q + q^2
sage: figure_eight.colored_jones_polynomial(3, 'Q')
Q^-6 - Q^-5 - Q^-4 + 2*Q^-3 - Q^-2 - Q^-1 + 3 - Q - Q^2
 + 2*Q^3 - Q^4 - Q^5 + Q^6
```

```
>>> from sage.all import *
>>> trefoil = BraidGroup(Integer(2))([Integer(1),Integer(1),Integer(1)])
>>> trefoil.colored_jones_polynomial(Integer(2))
q + q^3 - q^4
>>> trefoil.colored_jones_polynomial(Integer(4))
```

(continues on next page)

(continued from previous page)

```
q^3 + q^7 - q^10 + q^11 - q^13 - q^14 + q^15 - q^17
 + q^19 + q^20 - q^21
>>> trefoil.inverse().colored_jones_polynomial(Integer(4))
-q^-21 + q^-20 + q^-19 - q^-17 + q^-15 - q^-14 - q^-13
 + q^-11 - q^-10 + q^-7 + q^-3

>>> figure_eight = BraidGroup(Integer(3))([-Integer(1), Integer(2), -
↪Integer(1), Integer(2)])
>>> figure_eight.colored_jones_polynomial(Integer(2))
q^-2 - q^-1 + 1 - q + q^2
>>> figure_eight.colored_jones_polynomial(Integer(3), 'Q')
Q^-6 - Q^-5 - Q^-4 + 2*Q^-3 - Q^-2 - Q^-1 + 3 - Q - Q^2
 + 2*Q^3 - Q^4 - Q^5 + Q^6
```

components_in_closure()

Return the number of components of the trace closure of the braid.

OUTPUT: a positive integer

EXAMPLES:

```
sage: B = BraidGroup(5)
sage: b = B([1, -3])  # Three disjoint unknots
sage: b.components_in_closure()
3
sage: b = B([1, 2, 3, 4])  # The unknot
sage: b.components_in_closure()
1
sage: B = BraidGroup(4)
sage: K11n42 = B([1, -2, 3, -2, 3, -2, -2, -1, 2, -3, -3, 2, 2])
sage: K11n42.components_in_closure()
1
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(5))
>>> b = B([Integer(1), -Integer(3)])  # Three disjoint unknots
>>> b.components_in_closure()
3
>>> b = B([Integer(1), Integer(2), Integer(3), Integer(4)])  # The unknot
>>> b.components_in_closure()
1
>>> B = BraidGroup(Integer(4))
>>> K11n42 = B([Integer(1), -Integer(2), Integer(3), -Integer(2), Integer(3),␣
↪-Integer(2), -Integer(2), -Integer(1), Integer(2), -Integer(3), -Integer(3),
↪ Integer(2), Integer(2)])
>>> K11n42.components_in_closure()
1
```

conjugating_braid(*other*)

Return a conjugating braid, if it exists.

INPUT:

- `other` – a braid in the same braid group as `self`

OUTPUT:

A conjugating braid. More precisely, if the output is d, o equals `other`, and s equals `self` then $o = d^{-1} \cdot s \cdot d$.

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: B.one().conjugating_braid(B.one())
1
sage: B.one().conjugating_braid(B.gen(0)) is None
True
sage: B.gen(0).conjugating_braid(B.gen(1))
s1*s0
sage: B.gen(0).conjugating_braid(B.gen(1).inverse()) is None
True
sage: a = B([2, 2, -1, -1])
sage: b = B([2, 1, 2, 1])
sage: c = b * a / b
sage: d1 = a.conjugating_braid(c)
sage: d1
s1*s0
sage: d1 * c / d1 == a
True
sage: d1 * a / d1 == c
False
sage: l = sage.groups.braid.conjugatingbraid(a,c)                        #␣
↪needs sage.groups
sage: d1 == B._element_from_libbraiding(l)                               #␣
↪needs sage.groups
True
sage: b = B([2, 2, 2, 2, 1])
sage: c = b * a / b
sage: d1 = a.conjugating_braid(c)
sage: len(d1.Tietze())
7
sage: d1 * c / d1 == a
True
sage: d1 * a / d1 == c
False
sage: d1
s1^2*s0^2*s1^2*s0
sage: l = sage.groups.braid.conjugatingbraid(a,c)                        #␣
↪needs sage.groups
sage: d2 = B._element_from_libbraiding(l)                                #␣
↪needs sage.groups
sage: len(d2.Tietze())                                                   #␣
↪needs sage.groups
13
sage: c.conjugating_braid(b) is None
True
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> B.one().conjugating_braid(B.one())
```

(continues on next page)

(continued from previous page)

```
1
>>> B.one().conjugating_braid(B.gen(Integer(0))) is None
True
>>> B.gen(Integer(0)).conjugating_braid(B.gen(Integer(1)))
s1*s0
>>> B.gen(Integer(0)).conjugating_braid(B.gen(Integer(1)).inverse()) is None
True
>>> a = B([Integer(2), Integer(2), -Integer(1), -Integer(1)])
>>> b = B([Integer(2), Integer(1), Integer(2), Integer(1)])
>>> c = b * a / b
>>> d1 = a.conjugating_braid(c)
>>> d1
s1*s0
>>> d1 * c / d1 == a
True
>>> d1 * a / d1 == c
False
>>> l = sage.groups.braid.conjugatingbraid(a,c)                          #␣
↪needs sage.groups
>>> d1 == B._element_from_libbraiding(l)                                 #␣
↪needs sage.groups
True
>>> b = B([Integer(2), Integer(2), Integer(2), Integer(2), Integer(1)])
>>> c = b * a / b
>>> d1 = a.conjugating_braid(c)
>>> len(d1.Tietze())
7
>>> d1 * c / d1 == a
True
>>> d1 * a / d1 == c
False
>>> d1
s1^2*s0^2*s1^2*s0
>>> l = sage.groups.braid.conjugatingbraid(a,c)                          #␣
↪needs sage.groups
>>> d2 = B._element_from_libbraiding(l)                                  #␣
↪needs sage.groups
>>> len(d2.Tietze())                                                     #␣
↪needs sage.groups
13
>>> c.conjugating_braid(b) is None
True
```

cyclic_slidings()

Return the braid's cyclic slidings.

OUTPUT: The braid's cyclic slidings. Each cyclic sliding is a list of braids.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, 1, 2, 3, -1, 2, 1])
sage: b.cyclic_slidings()
```

(continues on next page)

(continued from previous page)

```
[[s0*s2*s1*s0*s1*s2, s0*s1*s2*s1*s0^2, s1*s0*s2^2*s1*s0],
 [s0*s1*s2*s1^2*s0, s0*s1*s2*s1*s0*s2, s1*s0*s2*s0*s1*s2]]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1), Integer(2), Integer(1), Integer(2), Integer(3), -
↪Integer(1), Integer(2), Integer(1)])
>>> b.cyclic_slidings()
[[s0*s2*s1*s0*s1*s2, s0*s1*s2*s1*s0^2, s1*s0*s2^2*s1*s0],
 [s0*s1*s2*s1^2*s0, s0*s1*s2*s1*s0*s2, s1*s0*s2*s0*s1*s2]]
```

deformed_burau_matrix(*variab='q'*)

Return the deformed Burau matrix of the braid.

INPUT:

- `variab` – variable (default: `q`); the variable in the resulting laurent polynomial, which is the base ring for the free algebra constructed

OUTPUT: a matrix with elements in the free algebra `self._algebra`

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, -3, -2, 3, 1])
sage: db = b.deformed_burau_matrix(); db
[                                ap_0*ap_5 ... bp_0*ap_1*cm_3*bp_4]
...
[                           bm_2*bm_3*cp_5 ...       bm_2*am_3*bp_4]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1), Integer(2), -Integer(3), -Integer(2), Integer(3),␣
↪Integer(1)])
>>> db = b.deformed_burau_matrix(); db
[                                ap_0*ap_5 ... bp_0*ap_1*cm_3*bp_4]
...
[                           bm_2*bm_3*cp_5 ...       bm_2*am_3*bp_4]
```

We check how this relates to the nondeformed Burau matrix:

```
sage: def subs_gen(gen, q):
....:     gen_str = str(gen)
....:     v = q if 'p' in gen_str else 1/q
....:     if 'b' in gen_str:
....:         return v
....:     elif 'a' in gen_str:
....:         return 1 - v
....:     else:
....:         return 1
sage: db_base = db.parent().base_ring()
sage: q = db_base.base_ring().gen()
sage: db_simp = db.subs({gen: subs_gen(gen, q)
....:                    for gen in db_base.gens()})
```

(continues on next page)

(continued from previous page)

```
sage: db_simp
[ (1-2*q+q^2)       (q-q^2)   (q-q^2+q^3)     (q^2-q^3)]
[       (1-q)             q             0             0]
[           0             0         (1-q)             q]
[      (q^-2)             0 -(q^-2-q^-1)     -(q^-1-1)]
sage: burau = b.burau_matrix(); burau
[1 - 2*t + t^2        t - t^2 t - t^2 + t^3      t^2 - t^3]
[        1 - t              t             0              0]
[            0              0         1 - t              t]
[         t^-2              0   -t^-2 + t^-1      -t^-1 + 1]
sage: t = burau.parent().base_ring().gen()
sage: burau.subs({t:q}).change_ring(db_base) == db_simp
True
```

```
>>> from sage.all import *
>>> def subs_gen(gen, q):
...     gen_str = str(gen)
...     v = q if 'p' in gen_str else Integer(1)/q
...     if 'b' in gen_str:
...         return v
...     elif 'a' in gen_str:
...         return Integer(1) - v
...     else:
...         return Integer(1)
>>> db_base = db.parent().base_ring()
>>> q = db_base.base_ring().gen()
>>> db_simp = db.subs({gen: subs_gen(gen, q)
...                    for gen in db_base.gens()})
>>> db_simp
[ (1-2*q+q^2)       (q-q^2)   (q-q^2+q^3)     (q^2-q^3)]
[       (1-q)             q             0             0]
[           0             0         (1-q)             q]
[      (q^-2)             0 -(q^-2-q^-1)     -(q^-1-1)]
>>> burau = b.burau_matrix(); burau
[1 - 2*t + t^2        t - t^2 t - t^2 + t^3      t^2 - t^3]
[        1 - t              t             0              0]
[            0              0         1 - t              t]
[         t^-2              0   -t^-2 + t^-1      -t^-1 + 1]
>>> t = burau.parent().base_ring().gen()
>>> burau.subs({t:q}).change_ring(db_base) == db_simp
True
```

gcd(*other*)

Return the greatest common divisor of the two braids.

INPUT:

- `other` – the other braid with respect with the gcd is computed

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: b = B([1, 2, -1, -2, -2, 1])
```

(continues on next page)

(continued from previous page)

```
sage: c = B([1, 2, 1])
sage: b.gcd(c)
s0^-1*s1^-1*s0^-2*s1^2*s0
sage: c.gcd(b)
s0^-1*s1^-1*s0^-2*s1^2*s0
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> b = B([Integer(1), Integer(2), -Integer(1), -Integer(2), -Integer(2),␣
→Integer(1)])
>>> c = B([Integer(1), Integer(2), Integer(1)])
>>> b.gcd(c)
s0^-1*s1^-1*s0^-2*s1^2*s0
>>> c.gcd(b)
s0^-1*s1^-1*s0^-2*s1^2*s0
```

is_conjugated(*other*)

Check if the two braids are conjugated.

INPUT:

- `other` – the other braid to check for conjugacy

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: a = B([2, 2, -1, -1])
sage: b = B([2, 1, 2, 1])
sage: c = b * a / b
sage: c.is_conjugated(a)
True
sage: c.is_conjugated(b)
False
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> a = B([Integer(2), Integer(2), -Integer(1), -Integer(1)])
>>> b = B([Integer(2), Integer(1), Integer(2), Integer(1)])
>>> c = b * a / b
>>> c.is_conjugated(a)
True
>>> c.is_conjugated(b)
False
```

is_periodic()

Check whether the braid is periodic.

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: a = B([2, 2, -1, -1, 2, 2])
sage: b = B([2, 1, 2, 1])
sage: a.is_periodic()
```

(continues on next page)

(continued from previous page)

```
False
sage: b.is_periodic()
True
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> a = B([Integer(2), Integer(2), -Integer(1), -Integer(1), Integer(2),␣
↪Integer(2)])
>>> b = B([Integer(2), Integer(1), Integer(2), Integer(1)])
>>> a.is_periodic()
False
>>> b.is_periodic()
True
```

is_pseudoanosov()

Check if the braid is pseudo-Anosov.

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: a = B([2, 2, -1, -1, 2, 2])
sage: b = B([2, 1, 2, 1])
sage: a.is_pseudoanosov()
True
sage: b.is_pseudoanosov()
False
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> a = B([Integer(2), Integer(2), -Integer(1), -Integer(1), Integer(2),␣
↪Integer(2)])
>>> b = B([Integer(2), Integer(1), Integer(2), Integer(1)])
>>> a.is_pseudoanosov()
True
>>> b.is_pseudoanosov()
False
```

is_reducible()

Check whether the braid is reducible.

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: b = B([1, 2, -1])
sage: b.is_reducible()
True
sage: a = B([2, 2, -1, -1, 2, 2])
sage: a.is_reducible()
False
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> b = B([Integer(1), Integer(2), -Integer(1)])
>>> b.is_reducible()
True
>>> a = B([Integer(2), Integer(2), -Integer(1), -Integer(1), Integer(2),␣
↪Integer(2)])
>>> a.is_reducible()
False
```

jones_polynomial(*variab=None*, *skein_normalization=False*)

Return the Jones polynomial of the trace closure of the braid.

The normalization is so that the unknot has Jones polynomial 1. If `skein_normalization` is `True`, the variable of the result is replaced by a itself to the power of 4, so that the result agrees with the conventions of [Lic1997] (which in particular differs slightly from the conventions used otherwise in this class), had one used the conventional Kauffman bracket variable notation directly.

If `variab` is `None` return a polynomial in the variable A or t, depending on the value `skein_normalization`. In particular, if `skein_normalization` is `False`, return the result in terms of the variable t, also used in [Lic1997].

INPUT:

- `variab` – variable (default: `None`); the variable in the resulting polynomial; if unspecified, use either a default variable in $\mathbf{Z}[A, A^{-1}]$ or the variable t in the symbolic ring
- `skein_normalization` – boolean (default: `False`); determines the variable of the resulting polynomial

OUTPUT:

If `skein_normalization` if `False`, this returns an element in the symbolic ring as the Jones polynomial of the closure might have fractional powers when the closure of the braid is not a knot. Otherwise the result is a Laurent polynomial in `variab`.

EXAMPLES:

The unknot:

```
sage: B = BraidGroup(9)
sage: b = B([1, 2, 3, 4, 5, 6, 7, 8])
sage: b.jones_polynomial()                                              #␣
↪needs sage.symbolic
1
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(9))
>>> b = B([Integer(1), Integer(2), Integer(3), Integer(4), Integer(5),␣
↪Integer(6), Integer(7), Integer(8)])
>>> b.jones_polynomial()                                              #␣
↪needs sage.symbolic
1
```

Two unlinked unknots:

```
sage: B = BraidGroup(2)
sage: b = B([])
```

(continues on next page)

(continued from previous page)

```
sage: b.jones_polynomial()                                                  #␣
↪needs sage.symbolic
-sqrt(t) - 1/sqrt(t)
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(2))
>>> b = B([])
>>> b.jones_polynomial()                                                    #␣
↪needs sage.symbolic
-sqrt(t) - 1/sqrt(t)
```

The Hopf link:

```
sage: B = BraidGroup(2)
sage: b = B([-1,-1])
sage: b.jones_polynomial()                                                  #␣
↪needs sage.symbolic
-1/sqrt(t) - 1/t^(5/2)
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(2))
>>> b = B([-Integer(1),-Integer(1)])
>>> b.jones_polynomial()                                                    #␣
↪needs sage.symbolic
-1/sqrt(t) - 1/t^(5/2)
```

Different representations of the trefoil and one of its mirror:

```
sage: # needs sage.symbolic
sage: B = BraidGroup(2)
sage: b = B([-1, -1, -1])
sage: b.jones_polynomial(skein_normalization=True)
-A^-16 + A^-12 + A^-4
sage: b.jones_polynomial()
1/t + 1/t^3 - 1/t^4
sage: B = BraidGroup(3)
sage: b = B([-1, -2, -1, -2])
sage: b.jones_polynomial(skein_normalization=True)
-A^-16 + A^-12 + A^-4
sage: R.<x> = LaurentPolynomialRing(GF(2))
sage: b.jones_polynomial(skein_normalization=True, variab=x)
x^-16 + x^-12 + x^-4
sage: B = BraidGroup(3)
sage: b = B([1, 2, 1, 2])
sage: b.jones_polynomial(skein_normalization=True)
A^4 + A^12 - A^16
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> B = BraidGroup(Integer(2))
>>> b = B([-Integer(1), -Integer(1), -Integer(1)])
>>> b.jones_polynomial(skein_normalization=True)
```

(continues on next page)

(continued from previous page)

```
-A^-16 + A^-12 + A^-4
>>> b.jones_polynomial()
1/t + 1/t^3 - 1/t^4
>>> B = BraidGroup(Integer(3))
>>> b = B([-Integer(1), -Integer(2), -Integer(1), -Integer(2)])
>>> b.jones_polynomial(skein_normalization=True)
-A^-16 + A^-12 + A^-4
>>> R = LaurentPolynomialRing(GF(Integer(2)), names=('x',)); (x,) = R._first_
↪ngens(1)
>>> b.jones_polynomial(skein_normalization=True, variab=x)
x^-16 + x^-12 + x^-4
>>> B = BraidGroup(Integer(3))
>>> b = B([Integer(1), Integer(2), Integer(1), Integer(2)])
>>> b.jones_polynomial(skein_normalization=True)
A^4 + A^12 - A^16
```

K11n42 (the mirror of the "Kinoshita-Terasaka" knot) and K11n34 (the mirror of the "Conway" knot):

```
sage: B = BraidGroup(4)
sage: b11n42 = B([1, -2, 3, -2, 3, -2, -2, -1, 2, -3, -3, 2, 2])
sage: b11n34 = B([1, 1, 2, -3, 2, -3, 1, -2, -2, -3, -3])
sage: bool(b11n42.jones_polynomial() == b11n34.jones_polynomial())          #␣
↪needs sage.symbolic
True
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b11n42 = B([Integer(1), -Integer(2), Integer(3), -Integer(2), Integer(3),␣
↪-Integer(2), -Integer(2), -Integer(1), Integer(2), -Integer(3), -Integer(3),
↪ Integer(2), Integer(2)])
>>> b11n34 = B([Integer(1), Integer(1), Integer(2), -Integer(3), Integer(2), -
↪Integer(3), Integer(1), -Integer(2), -Integer(2), -Integer(3), -Integer(3)])
>>> bool(b11n42.jones_polynomial() == b11n34.jones_polynomial())          #␣
↪needs sage.symbolic
True
```

lcm(*other*)

Return the least common multiple of the two braids.

INPUT:

- `other` – the other braid with respect with the lcm is computed

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: b = B([1, 2, -1, -2, -2, 1])
sage: c = B([1, 2, 1])
sage: b.lcm(c)
(s0*s1)^2*s0
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> b = B([Integer(1), Integer(2), -Integer(1), -Integer(2), -Integer(2),␣
→Integer(1)])
>>> c = B([Integer(1), Integer(2), Integer(1)])
>>> b.lcm(c)
(s0*s1)^2*s0
```

left_normal_form(*algorithm='libbraiding'*)

Return the left normal form of the braid.

INPUT:

- `algorithm` – string (default: `'artin'`); must be one of the following:
  - `'artin'` – the general method for Artin groups is used
  - `'libbraiding'` – the algorithm from the `libbraiding` package

OUTPUT:

A tuple of simple generators in the left normal form. The first element is a power of Δ, and the rest are elements of the natural section lift from the corresponding symmetric group.

EXAMPLES:

```
sage: B = BraidGroup(6)
sage: B.one().left_normal_form()
(1,)
sage: b = B([-2, 2, -4, -4, 4, -5, -1, 4, -1, 1])
sage: L1 = b.left_normal_form(); L1
(s0^-1*s1^-1*s0^-1*s2^-1*s1^-1*s0^-1*s3^-1*s2^-1*s1^-1*s0^-1*s4^-1*s3^-1*s2^-
→1*s1^-1*s0^-1,
 s0*s2*s1*s0*s3*s2*s1*s0*s4*s3*s2*s1,
 s3)
sage: L1 == b.left_normal_form()
True
sage: B([1]).left_normal_form(algorithm='artin')
(1, s0)
sage: B([-3]).left_normal_form(algorithm='artin')
(s0^-1*s1^-1*s0^-1*s2^-1*s1^-1*s0^-1*s3^-1*s2^-1*s1^-1*s0^-1*s4^-1*s3^-1*s2^-
→1*s1^-1*s0^-1,
 s0*s1*s2*s3*s4*s0*s1*s2*s3*s1*s2*s0*s1*s0)
sage: B = BraidGroup(3)
sage: B([1,2,-1]).left_normal_form()
(s0^-1*s1^-1*s0^-1, s1*s0, s0*s1)
sage: B([1,2,1]).left_normal_form()
(s0*s1*s0,)
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(6))
>>> B.one().left_normal_form()
(1,)
>>> b = B([-Integer(2), Integer(2), -Integer(4), -Integer(4), Integer(4), -
→Integer(5), -Integer(1), Integer(4), -Integer(1), Integer(1)])
>>> L1 = b.left_normal_form(); L1
(s0^-1*s1^-1*s0^-1*s2^-1*s1^-1*s0^-1*s3^-1*s2^-1*s1^-1*s0^-1*s4^-1*s3^-1*s2^-
```

(continues on next page)

(continued from previous page)

```
↪1*s1^-1*s0^-1,
 s0*s2*s1*s0*s3*s2*s1*s0*s4*s3*s2*s1,
 s3)
>>> L1 == b.left_normal_form()
True
>>> B([Integer(1)]).left_normal_form(algorithm='artin')
(1, s0)
>>> B([-Integer(3)]).left_normal_form(algorithm='artin')
(s0^-1*s1^-1*s0^-1*s2^-1*s1^-1*s0^-1*s3^-1*s2^-1*s1^-1*s0^-1*s4^-1*s3^-1*s2^-
↪1*s1^-1*s0^-1,
 s0*s1*s2*s3*s4*s0*s1*s2*s3*s1*s2*s0*s1*s0)
>>> B = BraidGroup(Integer(3))
>>> B([Integer(1),Integer(2),-Integer(1)]).left_normal_form()
(s0^-1*s1^-1*s0^-1, s1*s0, s0*s1)
>>> B([Integer(1),Integer(2),Integer(1)]).left_normal_form()
(s0*s1*s0,)
```

links_gould_matrix(*symbolics=False*)

Return the representation matrix of `self` according to the R-matrix representation being attached to the quantum superalgebra $\mathfrak{sl}_q(2|1)$.

See [MW2012], section 3 and references given there.

INPUT:

- `symbolics` – boolean (default: `False`); if set to `True` the coefficients will be contained in the symbolic ring. Per default they are elements of a quotient ring of a three variate Laurent polynomial ring.

OUTPUT: the representation matrix of `self` over the ring according to the choice of the keyword `symbolics` (see the corresponding explanation)

EXAMPLES:

```
sage: Hopf = BraidGroup(2)([-1, -1])
sage: HopfLG = Hopf.links_gould_matrix()
sage: HopfLG.dimensions()
(16, 16)
sage: HopfLG.base_ring()
Univariate Quotient Polynomial Ring in Yrbar
  over Multivariate Laurent Polynomial Ring in s0r, s1r
  over Integer Ring with modulus Yr^2 + s0r^2*s1r^2 - s0r^2 - s1r^2 + 1
sage: HopfLGs = Hopf.links_gould_matrix(symbolics=True)                    #␣
↪needs sage.symbolic
sage: HopfLGs.base_ring()                                                   #␣
↪needs sage.symbolic
Symbolic Ring
```

```
>>> from sage.all import *
>>> Hopf = BraidGroup(Integer(2))([-Integer(1), -Integer(1)])
>>> HopfLG = Hopf.links_gould_matrix()
>>> HopfLG.dimensions()
(16, 16)
>>> HopfLG.base_ring()
Univariate Quotient Polynomial Ring in Yrbar
```

(continues on next page)

(continued from previous page)

```
  over Multivariate Laurent Polynomial Ring in s0r, s1r
  over Integer Ring with modulus Yr^2 + s0r^2*s1r^2 - s0r^2 - s1r^2 + 1
>>> HopfLGs = Hopf.links_gould_matrix(symbolics=True)                    #␣
↪needs sage.symbolic
>>> HopfLGs.base_ring()                                                  #␣
↪needs sage.symbolic
Symbolic Ring
```

links_gould_polynomial(*varnames=None*, *use_symbolics=False*)

Return the Links-Gould polynomial of the closure of `self`.

See [MW2012], section 3 and references given there.

INPUT:

- `varnames` – string (default: `'t0, t1'`)

OUTPUT: a Laurent polynomial in the given variable names

EXAMPLES:

```
sage: Hopf = BraidGroup(2)([-1, -1])
sage: Hopf.links_gould_polynomial()
-1 + t1^-1 + t0^-1 - t0^-1*t1^-1
sage: _ == Hopf.links_gould_polynomial(use_symbolics=True)
True
sage: Hopf.links_gould_polynomial(varnames='a, b')
-1 + b^-1 + a^-1 - a^-1*b^-1
sage: _ == Hopf.links_gould_polynomial(varnames='a, b', use_symbolics=True)
True
```

```
>>> from sage.all import *
>>> Hopf = BraidGroup(Integer(2))([-Integer(1), -Integer(1)])
>>> Hopf.links_gould_polynomial()
-1 + t1^-1 + t0^-1 - t0^-1*t1^-1
>>> _ == Hopf.links_gould_polynomial(use_symbolics=True)
True
>>> Hopf.links_gould_polynomial(varnames='a, b')
-1 + b^-1 + a^-1 - a^-1*b^-1
>>> _ == Hopf.links_gould_polynomial(varnames='a, b', use_symbolics=True)
True
```

REFERENCES:

- [MW2012]

markov_trace(*variab=None*, *normalized=True*)

Return the Markov trace of the braid.

The normalization is so that in the underlying braid group representation, the eigenvalues of the standard generators of the braid group are 1 and $-A^4$.

INPUT:

- `variab` – variable (default: `None`); the variable in the resulting polynomial; if `None`, then use the variable A in $\mathbf{Z}[A, A^{-1}]$

- `normalized` – boolean (default: `True`); if specified to be `False`, return instead a rescaled Laurent polynomial version of the Markov trace

OUTPUT:

If `normalized` is `False`, return instead the Markov trace of the braid, normalized by a factor of $(A^2 + A^{-2})^n$. The result is then a Laurent polynomial in `variab`. Otherwise it is a quotient of Laurent polynomials in `variab`.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, -3])
sage: mt = b.markov_trace(); mt
A^4/(A^12 + 3*A^8 + 3*A^4 + 1)
sage: mt.factor()
A^4 * (A^4 + 1)^-3
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1), Integer(2), -Integer(3)])
>>> mt = b.markov_trace(); mt
A^4/(A^12 + 3*A^8 + 3*A^4 + 1)
>>> mt.factor()
A^4 * (A^4 + 1)^-3
```

We now give the non-normalized Markov trace:

```
sage: mt = b.markov_trace(normalized=False); mt
A^-4 + 1
sage: mt.parent()
Univariate Laurent Polynomial Ring in A over Integer Ring
```

```
>>> from sage.all import *
>>> mt = b.markov_trace(normalized=False); mt
A^-4 + 1
>>> mt.parent()
Univariate Laurent Polynomial Ring in A over Integer Ring
```

mirror_image()

Return the image of `self` under the mirror involution (see *BraidGroup_class.mirror_involution()*). The link closure of it is mirrored to the closure of `self` (see the example below of a positive amphicheiral knot).

EXAMPLES:

```
sage: B5 = BraidGroup(5)
sage: b  = B5((-1, 2, -3, -1, -3, 4, 2, -3, 2, 4, 2, -3)) # closure K12a_427
sage: bm = b.mirror_image(); bm
s0*s1^-1*s2*s0*s2*s3^-1*s1^-1*s2*s1^-1*s3^-1*s1^-1*s2
sage: bm.is_conjugated(b)
True
sage: bm.is_conjugated(~b)
False
```

```
>>> from sage.all import *
>>> B5 = BraidGroup(Integer(5))
>>> b  = B5((-Integer(1), Integer(2), -Integer(3), -Integer(1), -Integer(3),␣
↪Integer(4), Integer(2), -Integer(3), Integer(2), Integer(4), Integer(2), -
↪Integer(3))) # closure K12a_427
>>> bm = b.mirror_image(); bm
s0*s1^-1*s2*s0*s2*s3^-1*s1^-1*s2*s1^-1*s3^-1*s1^-1*s2
>>> bm.is_conjugated(b)
True
>>> bm.is_conjugated(~b)
False
```

permutation(*W=None*)

Return the permutation induced by the braid in its strands.

INPUT:

- `W` – (optional) the permutation group to project `self` to; the default is `self.parent().coxeter_group()`

OUTPUT: the image of `self` under the natural projection map to `W`

EXAMPLES:

```
sage: B.<s0,s1,s2> = BraidGroup()
sage: S = SymmetricGroup(4)
sage: b = s0*s1/s2/s1
sage: c0 = b.permutation(W=S); c0
(1,4,2)
sage: c1 = b.permutation(W=Permutations(4)); c1
[4, 1, 3, 2]
sage: c1 == b.permutation()
True
```

```
>>> from sage.all import *
>>> B = BraidGroup(names=('s0', 's1', 's2',)); (s0, s1, s2,) = B._first_
↪ngens(3)
>>> S = SymmetricGroup(Integer(4))
>>> b = s0*s1/s2/s1
>>> c0 = b.permutation(W=S); c0
(1,4,2)
>>> c1 = b.permutation(W=Permutations(Integer(4))); c1
[4, 1, 3, 2]
>>> c1 == b.permutation()
True
```

The canonical section from the symmetric group to the braid group (sending a permutation to its associated permutation braid) can be recovered:

```
sage: B(c0)
s0*s1*s2*s1
sage: B(c0) == B(c1)
True
```

```
>>> from sage.all import *
>>> B(c0)
s0*s1*s2*s1
>>> B(c0) == B(c1)
True
```

plot(*color='rainbow'*, *orientation='bottom-top'*, *gap=0.05*, *aspect_ratio=1*, *axes=False*, ***kwds*)

Plot the braid.

The following options are available:

- `color` – (default: `'rainbow'`) the color of the strands. Possible values are:
  - `'rainbow'`, uses `rainbow()` according to the number of strands.
  - a valid color name for `bezier_path()` and `line()`. Used for all strands.
  - a list or a tuple of colors for each individual strand.
- `orientation` – (default: `'bottom-top'`) determines how the braid is printed. The possible values are:
  - `'bottom-top'`, the braid is printed from bottom to top
  - `'top-bottom'`, the braid is printed from top to bottom
  - `'left-right'`, the braid is printed from left to right
- `gap` – floating point number (default: 0.05); determines the size of the gap left when a strand goes under another
- `aspect_ratio` – floating point number (default: `1`); the aspect ratio
- `**kwds` – other keyword options that are passed to `bezier_path()` and `line()`

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: a = B([2, 2, -1, -1])
sage: b = B([2, 1, 2, 1])
sage: c = b * a / b
sage: d = a.conjugating_braid(c)
sage: d * c / d == a
True
sage: d
s1*s0
sage: d * a / d == c
False
sage: B = BraidGroup(4, 's')
sage: b = B([1, 2, 3, 1, 2, 1])
sage: b.plot()                                                        #␣
↪needs sage.plot
Graphics object consisting of 30 graphics primitives
sage: b.plot(color=["red", "blue", "red", "blue"])                    #␣
↪needs sage.plot
Graphics object consisting of 30 graphics primitives

sage: B.<s,t> = BraidGroup(3)
sage: b = t^-1*s^2
```

(continues on next page)

(continued from previous page)

```
sage: b.plot(orientation='left-right', color='red')                    #␣
↪needs sage.plot
Graphics object consisting of 12 graphics primitives
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> a = B([Integer(2), Integer(2), -Integer(1), -Integer(1)])
>>> b = B([Integer(2), Integer(1), Integer(2), Integer(1)])
>>> c = b * a / b
>>> d = a.conjugating_braid(c)
>>> d * c / d == a
True
>>> d
s1*s0
>>> d * a / d == c
False
>>> B = BraidGroup(Integer(4), 's')
>>> b = B([Integer(1), Integer(2), Integer(3), Integer(1), Integer(2),␣
↪Integer(1)])
>>> b.plot()                                                           #␣
↪needs sage.plot
Graphics object consisting of 30 graphics primitives
>>> b.plot(color=["red", "blue", "red", "blue"])                       #␣
↪needs sage.plot
Graphics object consisting of 30 graphics primitives

>>> B = BraidGroup(Integer(3), names=('s', 't',)); (s, t,) = B._first_ngens(2)
>>> b = t**-Integer(1)*s**Integer(2)
>>> b.plot(orientation='left-right', color='red')                      #␣
↪needs sage.plot
Graphics object consisting of 12 graphics primitives
```

plot3d(*color='rainbow'*)

Plot the braid in 3d.

The following option is available:

- `color` – (default: `'rainbow'`) the color of the strands. Possible values are:
  - `'rainbow'`, uses `rainbow()` according to the number of strands.
  - a valid color name for `bezier3d()`. Used for all strands.
  - a list or a tuple of colors for each individual strand.

EXAMPLES:

```
sage: B = BraidGroup(4, 's')
sage: b = B([1, 2, 3, 1, 2, 1])
sage: b.plot3d()                                                       #␣
↪needs sage.plot sage.symbolic
Graphics3d Object
sage: b.plot3d(color='red')                                            #␣
↪needs sage.plot sage.symbolic
Graphics3d Object
```

(continues on next page)

(continued from previous page)

```
sage: b.plot3d(color=["red", "blue", "red", "blue"])                      #␣
↪needs sage.plot sage.symbolic
Graphics3d Object
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4), 's')
>>> b = B([Integer(1), Integer(2), Integer(3), Integer(1), Integer(2),␣
↪Integer(1)])
>>> b.plot3d()                                                          #␣
↪needs sage.plot sage.symbolic
Graphics3d Object
>>> b.plot3d(color='red')                                               #␣
↪needs sage.plot sage.symbolic
Graphics3d Object
>>> b.plot3d(color=["red", "blue", "red", "blue"])                      #␣
↪needs sage.plot sage.symbolic
Graphics3d Object
```

pure_conjugating_braid(*other*)

Return a pure conjugating braid, i.e. a conjugating braid whose associated permutation is the identity, if it exists.

INPUT:

- `other` – a braid in the same braid group as `self`

OUTPUT:

A pure conjugating braid. More precisely, if the output is d, o equals `other`, and s equals `self` then $o = d^{-1} \cdot s \cdot d$.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: B.one().pure_conjugating_braid(B.one())
1
sage: B.one().pure_conjugating_braid(B.gen(0)) is None
True
sage: B.gen(0).pure_conjugating_braid(B.gen(1)) is None
True
sage: B.gen(0).conjugating_braid(B.gen(2).inverse()) is None
True
sage: a = B([1, 2, 3])
sage: b = B([3, 2,])
sage: c = b ^ 12 * a / b ^ 12
sage: d1 = a.conjugating_braid(c)
sage: len(d1.Tietze())
30
sage: S = SymmetricGroup(4)
sage: d1.permutation(W=S)
(1,3)(2,4)
sage: d1 * c / d1 == a
True
sage: d1 * a / d1 == c
```

(continues on next page)

(continued from previous page)

```
False
sage: d2 = a.pure_conjugating_braid(c)
sage: len(d2.Tietze())
24
sage: d2.permutation(W=S)
()
sage: d2 * c / d2 == a
True
sage: d2
(s0*s1*s2^2*s1*s0)^4
sage: a.conjugating_braid(b) is None
True
sage: a.pure_conjugating_braid(b) is None
True
sage: a1 = B([1])
sage: a2 = B([2])
sage: a1.conjugating_braid(a2)
s1*s0
sage: a1.permutation(W=S)
(1,2)
sage: a2.permutation(W=S)
(2,3)
sage: a1.pure_conjugating_braid(a2) is None
True
sage: (a1^2).conjugating_braid(a2^2)
s1*s0
sage: (a1^2).pure_conjugating_braid(a2^2) is None
True
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> B.one().pure_conjugating_braid(B.one())
1
>>> B.one().pure_conjugating_braid(B.gen(Integer(0))) is None
True
>>> B.gen(Integer(0)).pure_conjugating_braid(B.gen(Integer(1))) is None
True
>>> B.gen(Integer(0)).conjugating_braid(B.gen(Integer(2)).inverse()) is None
True
>>> a = B([Integer(1), Integer(2), Integer(3)])
>>> b = B([Integer(3), Integer(2),])
>>> c = b ** Integer(12) * a / b ** Integer(12)
>>> d1 = a.conjugating_braid(c)
>>> len(d1.Tietze())
30
>>> S = SymmetricGroup(Integer(4))
>>> d1.permutation(W=S)
(1,3)(2,4)
>>> d1 * c / d1 == a
True
>>> d1 * a / d1 == c
False
```

(continues on next page)

(continued from previous page)

```
>>> d2 = a.pure_conjugating_braid(c)
>>> len(d2.Tietze())
24
>>> d2.permutation(W=S)
()
>>> d2 * c / d2 == a
True
>>> d2
(s0*s1*s2^2*s1*s0)^4
>>> a.conjugating_braid(b) is None
True
>>> a.pure_conjugating_braid(b) is None
True
>>> a1 = B([Integer(1)])
>>> a2 = B([Integer(2)])
>>> a1.conjugating_braid(a2)
s1*s0
>>> a1.permutation(W=S)
(1,2)
>>> a2.permutation(W=S)
(2,3)
>>> a1.pure_conjugating_braid(a2) is None
True
>>> (a1**Integer(2)).conjugating_braid(a2**Integer(2))
s1*s0
>>> (a1**Integer(2)).pure_conjugating_braid(a2**Integer(2)) is None
True
```

reverse()

Return the reverse of `self` obtained by reversing the order of the generators in its word. This defines an anti-involution on the braid group. The link closure of it has the reversed orientation (see the example below of a non reversible knot).

EXAMPLES:

```
sage: b  = BraidGroup(3)((1, 1, -2, 1, -2, 1, -2, -2))  # closure K8_17
sage: br = b.reverse(); br
s1^-1*(s1^-1*s0)^3*s0
sage: br.is_conjugated(b)
False
```

```
>>> from sage.all import *
>>> b  = BraidGroup(Integer(3))((Integer(1), Integer(1), -Integer(2),␣
↪Integer(1), -Integer(2), Integer(1), -Integer(2), -Integer(2)))  # closure␣
↪K8_17
>>> br = b.reverse(); br
s1^-1*(s1^-1*s0)^3*s0
>>> br.is_conjugated(b)
False
```

right_normal_form()

Return the right normal form of the braid.

A tuple of simple generators in the right normal form. The last element is a power of Δ, and the rest are elements of the natural section lift from the corresponding symmetric group.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, 1, -2, 3, 1])
sage: b.right_normal_form()
(s1*s0, s0*s2, 1)
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1), Integer(2), Integer(1), -Integer(2), Integer(3),␣
→Integer(1)])
>>> b.right_normal_form()
(s1*s0, s0*s2, 1)
```

rigidity()

Return the rigidity of `self`.

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: b = B([2, 1, 2, 1])
sage: a = B([2, 2, -1, -1, 2, 2])
sage: a.rigidity()
6
sage: b.rigidity()
0
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> b = B([Integer(2), Integer(1), Integer(2), Integer(1)])
>>> a = B([Integer(2), Integer(2), -Integer(1), -Integer(1), Integer(2),␣
→Integer(2)])
>>> a.rigidity()
6
>>> b.rigidity()
0
```

sliding_circuits()

Return the sliding circuits of the braid.

OUTPUT: list of sliding circuits. Each sliding circuit is itself a list of braids.

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: a = B([2, 2, -1, -1, 2, 2])
sage: a.sliding_circuits()
[[(s0^-1*s1^-1*s0^-1)^2*s1^3*s0^2*s1^3],
 [s0^-1*s1^-1*s0^-2*s1^-1*s0^2*s1^2*s0^3],
 [s0^-1*s1^-1*s0^-2*s1^-1*s0^3*s1^2*s0^2],
 [(s0^-1*s1^-1*s0^-1)^2*s1^4*s0^2*s1^2],
 [(s0^-1*s1^-1*s0^-1)^2*s1^2*s0^2*s1^4],
```

(continues on next page)

(continued from previous page)

```
 [s0^-1*s1^-1*s0^-2*s1^-1*s0*s1^2*s0^4],
 [(s0^-1*s1^-1*s0^-1)^2*s1^5*s0^2*s1],
 [s0^-1*s1^-1*s0^-2*s1^-1*s0^4*s1^2*s0],
 [(s0^-1*s1^-1*s0^-1)^2*s1*s0^2*s1^5],
 [s0^-1*s1^-1*s0^-2*s1*s0^5],
 [(s0^-1*s1^-1*s0^-1)^2*s1*s0^6*s1],
 [s0^-1*s1^-1*s0^-2*s1^5*s0]]
sage: b = B([2, 1, 2, 1])
sage: b.sliding_circuits()
[[s0*s1*s0^2, (s0*s1)^2]]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> a = B([Integer(2), Integer(2), -Integer(1), -Integer(1), Integer(2),␣
↪Integer(2)])
>>> a.sliding_circuits()
[[(s0^-1*s1^-1*s0^-1)^2*s1^3*s0^2*s1^3],
 [s0^-1*s1^-1*s0^-2*s1^-1*s0^2*s1^2*s0^3],
 [s0^-1*s1^-1*s0^-2*s1^-1*s0^3*s1^2*s0^2],
 [(s0^-1*s1^-1*s0^-1)^2*s1^4*s0^2*s1^2],
 [(s0^-1*s1^-1*s0^-1)^2*s1^2*s0^2*s1^4],
 [s0^-1*s1^-1*s0^-2*s1^-1*s0*s1^2*s0^4],
 [(s0^-1*s1^-1*s0^-1)^2*s1^5*s0^2*s1],
 [s0^-1*s1^-1*s0^-2*s1^-1*s0^4*s1^2*s0],
 [(s0^-1*s1^-1*s0^-1)^2*s1*s0^2*s1^5],
 [s0^-1*s1^-1*s0^-2*s1*s0^5],
 [(s0^-1*s1^-1*s0^-1)^2*s1*s0^6*s1],
 [s0^-1*s1^-1*s0^-2*s1^5*s0]]
>>> b = B([Integer(2), Integer(1), Integer(2), Integer(1)])
>>> b.sliding_circuits()
[[s0*s1*s0^2, (s0*s1)^2]]
```

sliding_circuits_element()

Return an element of the braid's sliding circuits, and the conjugating braid.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, 1, 2, 3, -1, 2, -1, 3])
sage: b.sliding_circuits_element()
(s0*s1*s0*s2*s1, s0^2*s1*s2)
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1), Integer(2), Integer(1), Integer(2), Integer(3), -
↪Integer(1), Integer(2), -Integer(1), Integer(3)])
>>> b.sliding_circuits_element()
(s0*s1*s0*s2*s1, s0^2*s1*s2)
```

strands()

Return the number of strands in the braid.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, -1, 3, -2])
sage: b.strands()
4
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1), Integer(2), -Integer(1), Integer(3), -Integer(2)])
>>> b.strands()
4
```

super_summit_set()

Return a list with the super summit set of the braid.

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: b = B([1, 2, -1, -2, -2, 1])
sage: b.super_summit_set()
[s0^-1*s1^-1*s0^-2*s1^2*s0^2,
 (s0^-1*s1^-1*s0^-1)^2*s1^2*s0^3*s1,
 (s0^-1*s1^-1*s0^-1)^2*s1*s0^3*s1^2,
 s0^-1*s1^-1*s0^-2*s1^-1*s0*s1^3*s0]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> b = B([Integer(1), Integer(2), -Integer(1), -Integer(2), -Integer(2),␣
↪Integer(1)])
>>> b.super_summit_set()
[s0^-1*s1^-1*s0^-2*s1^2*s0^2,
 (s0^-1*s1^-1*s0^-1)^2*s1^2*s0^3*s1,
 (s0^-1*s1^-1*s0^-1)^2*s1*s0^3*s1^2,
 s0^-1*s1^-1*s0^-2*s1^-1*s0*s1^3*s0]
```

super_summit_set_element()

Return an element of the braid's super summit set and the conjugating braid.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, 1, 2, 3, -1, 2, 1, 3])
sage: b.super_summit_set_element()
(s0*s2*s0*s1*s2*s1*s0, s0^-1*s1^-1*s0^-1*s2^-1*s1^-1*s0^-1*s1*s0*s2*s1*s0)
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1), Integer(2), Integer(1), Integer(2), Integer(3), -
↪Integer(1), Integer(2), Integer(1), Integer(3)])
>>> b.super_summit_set_element()
(s0*s2*s0*s1*s2*s1*s0, s0^-1*s1^-1*s0^-1*s2^-1*s1^-1*s0^-1*s1*s0*s2*s1*s0)
```

thurston_type()

Return the thurston_type of `self`.

OUTPUT: one of `'reducible'`, `'periodic'` or `'pseudo-anosov'`

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: b = B([1, 2, -1])
sage: b.thurston_type()
'reducible'
sage: a = B([2, 2, -1, -1, 2, 2])
sage: a.thurston_type()
'pseudo-anosov'
sage: c = B([2, 1, 2, 1])
sage: c.thurston_type()
'periodic'
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> b = B([Integer(1), Integer(2), -Integer(1)])
>>> b.thurston_type()
'reducible'
>>> a = B([Integer(2), Integer(2), -Integer(1), -Integer(1), Integer(2),␣
↪Integer(2)])
>>> a.thurston_type()
'pseudo-anosov'
>>> c = B([Integer(2), Integer(1), Integer(2), Integer(1)])
>>> c.thurston_type()
'periodic'
```

trajectory()

Return the braid's trajectory.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, 1, 2, 3, -1, 2, -1, 3])
sage: b.trajectory()
[s0^-1*s1^-1*s0^-1*s2^-1*s1^-1*s2*s0*s1*s2*s1*s0^2*s1*s2^2,
 s0*s1*s2^3,
 s0*s1*s2*s1^2,
 s0*s1*s0*s2*s1]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1), Integer(2), Integer(1), Integer(2), Integer(3), -
↪Integer(1), Integer(2), -Integer(1), Integer(3)])
>>> b.trajectory()
[s0^-1*s1^-1*s0^-1*s2^-1*s1^-1*s2*s0*s1*s2*s1*s0^2*s1*s2^2,
 s0*s1*s2^3,
 s0*s1*s2*s1^2,
 s0*s1*s0*s2*s1]
```

tropical_coordinates()

Return the tropical coordinates of `self` in the braid group B_n.

OUTPUT: list of $2n$ tropical integers

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: b = B([1])
sage: tc = b.tropical_coordinates(); tc
[1, 0, 0, 2, 0, 1]
sage: tc[0].parent()
Tropical semiring over Integer Ring

sage: b = B([-2, -2, -1, -1, 2, 2, 1, 1])
sage: b.tropical_coordinates()
[1, -19, -12, 9, 0, 13]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> b = B([Integer(1)])
>>> tc = b.tropical_coordinates(); tc
[1, 0, 0, 2, 0, 1]
>>> tc[Integer(0)].parent()
Tropical semiring over Integer Ring

>>> b = B([-Integer(2), -Integer(2), -Integer(1), -Integer(1), Integer(2),␣
↪Integer(2), Integer(1), Integer(1)])
>>> b.tropical_coordinates()
[1, -19, -12, 9, 0, 13]
```

REFERENCES:

- [DW2007]
- [Deh2011]

ultra_summit_set()

Return a list with the orbits of the ultra summit set of `self`.

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: a = B([2, 2, -1, -1, 2, 2])
sage: b = B([2, 1, 2, 1])
sage: b.ultra_summit_set()
[[s0*s1*s0^2, (s0*s1)^2]]
sage: a.ultra_summit_set()
[[(s0^-1*s1^-1*s0^-1)^2*s1^3*s0^2*s1^3,
  (s0^-1*s1^-1*s0^-1)^2*s1^2*s0^2*s1^4,
  (s0^-1*s1^-1*s0^-1)^2*s1*s0^2*s1^5,
  s0^-1*s1^-1*s0^-2*s1^5*s0,
  (s0^-1*s1^-1*s0^-1)^2*s1^5*s0^2*s1,
  (s0^-1*s1^-1*s0^-1)^2*s1^4*s0^2*s1^2],
 [s0^-1*s1^-1*s0^-2*s1^-1*s0^2*s1^2*s0^3,
  s0^-1*s1^-1*s0^-2*s1^-1*s0*s1^2*s0^4,
  s0^-1*s1^-1*s0^-2*s1*s0^5,
  (s0^-1*s1^-1*s0^-1)^2*s1*s0^6*s1,
  s0^-1*s1^-1*s0^-2*s1^-1*s0^4*s1^2*s0,
  s0^-1*s1^-1*s0^-2*s1^-1*s0^3*s1^2*s0^2]]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> a = B([Integer(2), Integer(2), -Integer(1), -Integer(1), Integer(2),␣
↪Integer(2)])
>>> b = B([Integer(2), Integer(1), Integer(2), Integer(1)])
>>> b.ultra_summit_set()
[[s0*s1*s0^2, (s0*s1)^2]]
>>> a.ultra_summit_set()
[[(s0^-1*s1^-1*s0^-1)^2*s1^3*s0^2*s1^3,
  (s0^-1*s1^-1*s0^-1)^2*s1^2*s0^2*s1^4,
  (s0^-1*s1^-1*s0^-1)^2*s1*s0^2*s1^5,
  s0^-1*s1^-1*s0^-2*s1^5*s0,
  (s0^-1*s1^-1*s0^-1)^2*s1^5*s0^2*s1,
  (s0^-1*s1^-1*s0^-1)^2*s1^4*s0^2*s1^2],
 [s0^-1*s1^-1*s0^-2*s1^-1*s0^2*s1^2*s0^3,
  s0^-1*s1^-1*s0^-2*s1^-1*s0*s1^2*s0^4,
  s0^-1*s1^-1*s0^-2*s1*s0^5,
  (s0^-1*s1^-1*s0^-1)^2*s1*s0^6*s1,
  s0^-1*s1^-1*s0^-2*s1^-1*s0^4*s1^2*s0,
  s0^-1*s1^-1*s0^-2*s1^-1*s0^3*s1^2*s0^2]]
```

ultra_summit_set_element()

Return an element of the braid's ultra summit set and the conjugating braid.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, 1, 2, 3, -1, 2, -1, 3])
sage: b.ultra_summit_set_element()
(s0*s1*s0*s2*s1, s0^-1*s1^-1*s0^-1*s2^-1*s1^-1*s0^-1*s1*s2*s1^2*s0)
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1), Integer(2), Integer(1), Integer(2), Integer(3), -
↪Integer(1), Integer(2), -Integer(1), Integer(3)])
>>> b.ultra_summit_set_element()
(s0*s1*s0*s2*s1, s0^-1*s1^-1*s0^-1*s2^-1*s1^-1*s0^-1*s1*s2*s1^2*s0)
```

sage.groups.braid.**BraidGroup**(*n=None*, *names='s'*)

Construct a Braid Group.

INPUT:

- n – integer or None (default). The number of strands. If not specified the names are counted and the group is assumed to have one more strand than generators.
- names – string or list/tuple/iterable of strings (default: 'x'); the generator names or name prefix

EXAMPLES:

```
sage: B.<a,b> = BraidGroup();  B
Braid group on 3 strands
sage: H = BraidGroup('a, b')
sage: B is H
True
```

(continues on next page)

(continued from previous page)

```
sage: BraidGroup(3)
Braid group on 3 strands
```

```
>>> from sage.all import *
>>> B = BraidGroup(names=('a', 'b',)); (a, b,) = B._first_ngens(2);  B
Braid group on 3 strands
>>> H = BraidGroup('a, b')
>>> B is H
True
>>> BraidGroup(Integer(3))
Braid group on 3 strands
```

The entry can be either a string with the names of the generators, or the number of generators and the prefix of the names to be given. The default prefix is `'s'`

```
sage: B = BraidGroup(3); B.generators()
(s0, s1)
sage: BraidGroup(3, 'g').generators()
(g0, g1)
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3)); B.generators()
(s0, s1)
>>> BraidGroup(Integer(3), 'g').generators()
(g0, g1)
```

Since the word problem for the braid groups is solvable, their Cayley graph can be locally obtained as follows (see Issue #16059):

```
sage: def ball(group, radius):
....:     ret = set()
....:     ret.add(group.one())
....:     for length in range(1, radius):
....:         for w in Words(alphabet=group.gens(), length=length):
....:              ret.add(prod(w))
....:     return ret
sage: B = BraidGroup(4)
sage: GB = B.cayley_graph(elements=ball(B, 4), generators=B.gens()); GB          #␣
↪needs sage.combinat sage.graphs
Digraph on 31 vertices
```

```
>>> from sage.all import *
>>> def ball(group, radius):
...     ret = set()
...     ret.add(group.one())
...     for length in range(Integer(1), radius):
...         for w in Words(alphabet=group.gens(), length=length):
...              ret.add(prod(w))
...     return ret
>>> B = BraidGroup(Integer(4))
>>> GB = B.cayley_graph(elements=ball(B, Integer(4)), generators=B.gens()); GB   ␣
```

(continues on next page)

(continued from previous page)

```
↪     # needs sage.combinat sage.graphs
Digraph on 31 vertices
```

Since the braid group has nontrivial relations, this graph contains less vertices than the one associated to the free group (which is a tree):

```
sage: F = FreeGroup(3)
sage: GF = F.cayley_graph(elements=ball(F, 4), generators=F.gens()); GF          #␣
↪needs sage.combinat sage.graphs
Digraph on 40 vertices
```

```
>>> from sage.all import *
>>> F = FreeGroup(Integer(3))
>>> GF = F.cayley_graph(elements=ball(F, Integer(4)), generators=F.gens()); GF    ␣
↪     # needs sage.combinat sage.graphs
Digraph on 40 vertices
```

class sage.groups.braid.**BraidGroup_class**(*names*)

Bases: *FiniteTypeArtinGroup*

The braid group on n strands.

EXAMPLES:

```
sage: B1 = BraidGroup(5)
sage: B1
Braid group on 5 strands
sage: B2 = BraidGroup(3)
sage: B1==B2
False
sage: B2 is BraidGroup(3)
True
```

```
>>> from sage.all import *
>>> B1 = BraidGroup(Integer(5))
>>> B1
Braid group on 5 strands
>>> B2 = BraidGroup(Integer(3))
>>> B1==B2
False
>>> B2 is BraidGroup(Integer(3))
True
```

Element

alias of *Braid*

TL_basis_with_drain(*drain_size*)

Return a basis of a summand of the Temperley–Lieb–Jones representation of `self`.

The basis elements are given by non-intersecting pairings of $n + d$ points in a square with n points marked 'on the top' and d points 'on the bottom' so that every bottom point is paired with a top point. Here, n is the number of strands of the braid group, and d is specified by `drain_size`.

A basis element is specified as a list of integers obtained by considering the pairings as obtained as the 'highest term' of trivalent trees marked by Jones–Wenzl projectors (see e.g. [Wan2010]). In practice, this is a list of nonnegative integers whose first element is `drain_size`, whose last element is 0, and satisfying that consecutive integers have difference 1. Moreover, the length of each basis element is $n + 1$.

Given these rules, the list of lists is constructed recursively in the natural way.

INPUT:

- `drain_size` – integer between 0 and the number of strands (both inclusive)

OUTPUT: list of basis elements, each of which is a list of integers

EXAMPLES:

We calculate the basis for the appropriate vector space for B_5 when $d = 3$:

```
sage: B = BraidGroup(5)
sage: B.TL_basis_with_drain(3)
[[3, 4, 3, 2, 1, 0],
 [3, 2, 3, 2, 1, 0],
 [3, 2, 1, 2, 1, 0],
 [3, 2, 1, 0, 1, 0]]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(5))
>>> B.TL_basis_with_drain(Integer(3))
[[3, 4, 3, 2, 1, 0],
 [3, 2, 3, 2, 1, 0],
 [3, 2, 1, 2, 1, 0],
 [3, 2, 1, 0, 1, 0]]
```

The number of basis elements hopefully corresponds to the general formula for the dimension of the representation spaces:

```
sage: B = BraidGroup(10)
sage: d = 2
sage: B.dimension_of_TL_space(d) == len(B.TL_basis_with_drain(d))
True
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(10))
>>> d = Integer(2)
>>> B.dimension_of_TL_space(d) == len(B.TL_basis_with_drain(d))
True
```

TL_representation(*drain_size*, *variab=None*)

Return representation matrices of the Temperley–Lieb–Jones representation of standard braid group generators and inverses of `self`.

The basis is given by non-intersecting pairings of $(n + d)$ points, where n is the number of strands, and d is given by `drain_size`, and the pairings satisfy certain rules. See *TL_basis_with_drain()* for details. This basis has the useful property that all resulting entries can be regarded as Laurent polynomials.

We use the convention that the eigenvalues of the standard generators are 1 and $-A^4$, where A is the generator of the Laurent polynomial ring.

When $d = n - 2$ and the variables are picked appropriately, the resulting representation is equivalent to the reduced Burau representation. When $d = n$, the resulting representation is trivial and 1-dimensional.

INPUT:

- `drain_size` – integer between 0 and the number of strands (both inclusive)
- `variab` – variable (default: `None`); the variable in the entries of the matrices; if `None`, then use a default variable in $\mathbf{Z}[A, A^{-1}]$

OUTPUT: list of matrices corresponding to the representations of each of the standard generators and their inverses

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: B.TL_representation(0)
[(
  [   1     0]  [    1      0]
  [ A^2 -A^4], [ A^-2 -A^-4]
),
 (
  [-A^4  A^2]  [-A^-4  A^-2]
  [   0    1], [    0     1]
),
 (
  [   1     0]  [    1      0]
  [ A^2 -A^4], [ A^-2 -A^-4]
)]
sage: R.<A> = LaurentPolynomialRing(GF(2))
sage: B.TL_representation(0, variab=A)
[(
  [  1   0]  [   1    0]
  [A^2 A^4], [A^-2 A^-4]
),
 (
  [A^4 A^2]  [A^-4 A^-2]
  [  0   1], [   0    1]
),
 (
  [  1   0]  [   1    0]
  [A^2 A^4], [A^-2 A^-4]
)]
sage: B = BraidGroup(8)
sage: B.TL_representation(8)
[([1], [1]),
 ([1], [1]),
 ([1], [1]),
 ([1], [1]),
 ([1], [1]),
 ([1], [1]),
 ([1], [1])]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> B.TL_representation(Integer(0))
[(
  [   1     0]  [    1      0]
  [ A^2 -A^4], [ A^-2 -A^-4]
),
```

(continues on next page)

(continued from previous page)

```
 (
  [-A^4  A^2]  [-A^-4  A^-2]
  [   0    1], [    0     1]
),
 (
  [   1    0]  [    1     0]
  [ A^2 -A^4], [ A^-2 -A^-4]
)]
>>> R = LaurentPolynomialRing(GF(Integer(2)), names=('A',)); (A,) = R._first_
↪ngens(1)
>>> B.TL_representation(Integer(0), variab=A)
[(
  [  1   0]  [   1    0]
  [A^2 A^4], [A^-2 A^-4]
),
 (
  [A^4 A^2]  [A^-4 A^-2]
  [  0   1], [   0    1]
),
 (
  [  1   0]  [   1    0]
  [A^2 A^4], [A^-2 A^-4]
)]
>>> B = BraidGroup(Integer(8))
>>> B.TL_representation(Integer(8))
[([1], [1]),
 ([1], [1]),
 ([1], [1]),
 ([1], [1]),
 ([1], [1]),
 ([1], [1]),
 ([1], [1])]
```

an_element()

Return an element of the braid group.

This is used both for illustration and testing purposes.

EXAMPLES:

```
sage: B = BraidGroup(2)
sage: B.an_element()
s
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(2))
>>> B.an_element()
s
```

as_permutation_group()

Return an isomorphic permutation group.

OUTPUT: this raises a `ValueError` error since braid groups are infinite

cardinality()

Return the number of group elements.

OUTPUT: Infinity

dimension_of_TL_space(*drain_size*)

Return the dimension of a particular Temperley–Lieb representation summand of `self`.

Following the notation of *TL_basis_with_drain()*, the summand is the one corresponding to the number of drains being fixed to be `drain_size`.

INPUT:

- `drain_size` – integer between 0 and the number of strands (both inclusive)

EXAMPLES:

Calculation of the dimension of the representation of B_8 corresponding to having 2 drains:

```
sage: B = BraidGroup(8)
sage: B.dimension_of_TL_space(2)
28
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(8))
>>> B.dimension_of_TL_space(Integer(2))
28
```

The direct sum of endomorphism spaces of these vector spaces make up the entire Temperley–Lieb algebra:

```
sage: import sage.combinat.diagram_algebras as da                    #␣
↪needs sage.combinat
sage: B = BraidGroup(6)
sage: dimensions = [B.dimension_of_TL_space(d)**2 for d in [0, 2, 4, 6]]
sage: total_dim = sum(dimensions)
sage: total_dim == len(list(da.temperley_lieb_diagrams(6)))          # long␣
↪time, needs sage.combinat
True
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da                      #␣
↪needs sage.combinat
>>> B = BraidGroup(Integer(6))
>>> dimensions = [B.dimension_of_TL_space(d)**Integer(2) for d in [Integer(0),
↪ Integer(2), Integer(4), Integer(6)]]
>>> total_dim = sum(dimensions)
>>> total_dim == len(list(da.temperley_lieb_diagrams(Integer(6))))    #␣
↪long time, needs sage.combinat
True
```

epimorphisms(H)

Return the epimorphisms from `self` to `H`, up to automorphism of H passing through the *two generator presentation* of `self`.

INPUT:

- `H` – another group

EXAMPLES:

```
sage: B = BraidGroup(5)
sage: B.epimorphisms(SymmetricGroup(5))
[Generic morphism:
From: Braid group on 5 strands
To:   Symmetric group of order 5! as a permutation group
Defn: s0 |--> (1,5)
      s1 |--> (4,5)
      s2 |--> (3,4)
      s3 |--> (2,3)]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(5))
>>> B.epimorphisms(SymmetricGroup(Integer(5)))
[Generic morphism:
From: Braid group on 5 strands
To:   Symmetric group of order 5! as a permutation group
Defn: s0 |--> (1,5)
      s1 |--> (4,5)
      s2 |--> (3,4)
      s3 |--> (2,3)]
```

ALGORITHM:

Uses libgap's GQuotients function.

mapping_class_action(F)

Return the action of `self` in the free group F as mapping class group.

This action corresponds to the action of the braid over the punctured disk, whose fundamental group is the free group on as many generators as strands.

In Sage, this action is the result of multiplying a free group element with a braid. So you generally do not have to construct this action yourself.

OUTPUT: a *MappingClassGroupAction*

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: B.inject_variables()
Defining s0, s1
sage: F.<a,b,c> = FreeGroup(3)
sage: A = B.mapping_class_action(F)
sage: A(a,s0)
a*b*a^-1
sage: a * s0    # simpler notation
a*b*a^-1
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> B.inject_variables()
Defining s0, s1
>>> F = FreeGroup(Integer(3), names=('a', 'b', 'c',)); (a, b, c,) = F._first_
↪ngens(3)
```

(continues on next page)

(continued from previous page)

```
>>> A = B.mapping_class_action(F)
>>> A(a,s0)
a*b*a^-1
>>> a * s0     # simpler notation
a*b*a^-1
```

mirror_involution()

Return the mirror involution of `self`.

This automorphism maps a braid to another one by replacing each generator in its word by the inverse. In general this is different from the inverse of the braid since the order of the generators in the word is not reversed.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: mirr = B.mirror_involution()
sage: b = B((1,-2,-1,3,2,1))
sage: bm = mirr(b); bm
s0^-1*s1*s0*s2^-1*s1^-1*s0^-1
sage: bm == ~b
False
sage: bm.is_conjugated(b)
False
sage: bm.is_conjugated(~b)
True
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> mirr = B.mirror_involution()
>>> b = B((Integer(1),-Integer(2),-Integer(1),Integer(3),Integer(2),
↪Integer(1)))
>>> bm = mirr(b); bm
s0^-1*s1*s0*s2^-1*s1^-1*s0^-1
>>> bm == ~b
False
>>> bm.is_conjugated(b)
False
>>> bm.is_conjugated(~b)
True
```

order()

Return the number of group elements.

OUTPUT: Infinity

presentation_two_generators(*isomorphisms=False*)

Construct a finitely presented group isomorphic to `self` with only two generators.

INPUT:

- `isomorphism` – boolean (default: `False`); if `True`, then an isomorphism from `self` and the isomorphic group and its inverse is also returned

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: B.presentation_two_generators()
Finitely presented group < x0, x1 | x1^3*x0^-2 >
sage: B = BraidGroup(4)
sage: G, hom1, hom2 = B.presentation_two_generators(isomorphisms=True)
sage: G
Finitely presented group < x0, x1 | x1^4*x0^-3, x0*x1*x0*x1^-2*x0^-1*x1^3*x0^-
↪1*x1^-2 >
sage: hom1(B.gen(0))
x0*x1^-1
sage: hom1(B.gen(1))
x1*x0*x1^-2
sage: hom1(B.gen(2))
x1^2*x0*x1^-3
sage: all(hom2(hom1(a)) == a for a in B.gens())
True
sage: all(hom2(a) == B.one() for a in G.relations())
True
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> B.presentation_two_generators()
Finitely presented group < x0, x1 | x1^3*x0^-2 >
>>> B = BraidGroup(Integer(4))
>>> G, hom1, hom2 = B.presentation_two_generators(isomorphisms=True)
>>> G
Finitely presented group < x0, x1 | x1^4*x0^-3, x0*x1*x0*x1^-2*x0^-1*x1^3*x0^-
↪1*x1^-2 >
>>> hom1(B.gen(Integer(0)))
x0*x1^-1
>>> hom1(B.gen(Integer(1)))
x1*x0*x1^-2
>>> hom1(B.gen(Integer(2)))
x1^2*x0*x1^-3
>>> all(hom2(hom1(a)) == a for a in B.gens())
True
>>> all(hom2(a) == B.one() for a in G.relations())
True
```

some_elements()

Return a list of some elements of the braid group.

This is used both for illustration and testing purposes.

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: B.some_elements()
[s0, s0*s1, (s0*s1)^3]
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> B.some_elements()
```

(continues on next page)

(continued from previous page)

```
[s0, s0*s1, (s0*s1)^3]
```

strands()

Return the number of strands.

OUTPUT: integer

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: B.strands()
4
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> B.strands()
4
```

class sage.groups.braid.**MappingClassGroupAction**(G, M)

Bases: Action

The right action of the braid group the free group as the mapping class group of the punctured disk.

That is, this action is the action of the braid over the punctured disk, whose fundamental group is the free group on as many generators as strands.

This action is defined as follows:

$$x_j \cdot \sigma_i = \begin{cases} x_j \cdot x_{j+1} \cdot {x_j}^{-1} & \text{if } i = j \\ x_{j-1} & \text{if } i = j-1 \\ x_j & \text{otherwise} \end{cases},$$

where σ_i are the generators of the braid group on n strands, and x_j the generators of the free group of rank n.

You should left multiplication of the free group element by the braid to compute the action. Alternatively, use the *mapping_class_action()* method of the braid group to construct this action.

EXAMPLES:

```
sage: B.<s0,s1,s2> = BraidGroup(4)
sage: F.<x0,x1,x2,x3> = FreeGroup(4)
sage: x0 * s1
x0
sage: x1 * s1
x1*x2*x1^-1
sage: x1^-1 * s1
x1*x2^-1*x1^-1

sage: A = B.mapping_class_action(F)
sage: A
Right action by Braid group on 4 strands on Free Group
on generators {x0, x1, x2, x3}
sage: A(x0, s1)
x0
sage: A(x1, s1)
```

(continues on next page)

(continued from previous page)

```
x1*x2*x1^-1
sage: A(x1^-1, s1)
x1*x2^-1*x1^-1
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4), names=('s0', 's1', 's2',)); (s0, s1, s2,) = B._
↪first_ngens(3)
>>> F = FreeGroup(Integer(4), names=('x0', 'x1', 'x2', 'x3',)); (x0, x1, x2, x3,)␣
↪= F._first_ngens(4)
>>> x0 * s1
x0
>>> x1 * s1
x1*x2*x1^-1
>>> x1**-Integer(1) * s1
x1*x2^-1*x1^-1

>>> A = B.mapping_class_action(F)
>>> A
Right action by Braid group on 4 strands on Free Group
on generators {x0, x1, x2, x3}
>>> A(x0, s1)
x0
>>> A(x1, s1)
x1*x2*x1^-1
>>> A(x1**-Integer(1), s1)
x1*x2^-1*x1^-1
```

class sage.groups.braid.**RightQuantumWord**(*words*)

Bases: `object`

A right quantum word as in Definition 4.1 of [HL2018].

INPUT:

- `words` – an element in a suitable free algebra over a Laurent polynomial ring in one variable; this input does not need to be in reduced form, but the monomials for the input can come in any order

EXAMPLES:

```
sage: from sage.groups.braid import RightQuantumWord
sage: fig_8 = BraidGroup(3)([-1, 2, -1, 2])
sage: (
....:  bp_1, cp_1, ap_1,
....:  bp_3, cp_3, ap_3,
....:  bm_0, cm_0, am_0,
....:  bm_2, cm_2, am_2
....: ) = fig_8.deformed_burau_matrix().parent().base_ring().gens()
sage: q = bp_1.base_ring().gen()
sage: RightQuantumWord(ap_1*cp_1 + q**3*bm_2*bp_1*am_0*cm_0)
The right quantum word represented by
 q*cp_1*ap_1 + q^2*bp_1*cm_0*am_0*bm_2
 reduced from ap_1*cp_1 + q^3*bm_2*bp_1*am_0*cm_0
```

```
>>> from sage.all import *
>>> from sage.groups.braid import RightQuantumWord
>>> fig_8 = BraidGroup(Integer(3))([-Integer(1), Integer(2), -Integer(1),␣
↪Integer(2)])
>>> (
...  bp_1, cp_1, ap_1,
...  bp_3, cp_3, ap_3,
...  bm_0, cm_0, am_0,
...  bm_2, cm_2, am_2
... ) = fig_8.deformed_burau_matrix().parent().base_ring().gens()
>>> q = bp_1.base_ring().gen()
>>> RightQuantumWord(ap_1*cp_1 + q**Integer(3)*bm_2*bp_1*am_0*cm_0)
The right quantum word represented by
 q*cp_1*ap_1 + q^2*bp_1*cm_0*am_0*bm_2
 reduced from ap_1*cp_1 + q^3*bm_2*bp_1*am_0*cm_0
```

eps (N)

Evaluate the map $\mathcal{E}_N$ for a braid.

INPUT:

- N – integer; the number of colors

EXAMPLES:

```
sage: from sage.groups.braid import RightQuantumWord
sage: B = BraidGroup(3)
sage: b = B([1,-2,1,2])
sage: db = b.deformed_burau_matrix()
sage: q = db.parent().base_ring().base_ring().gen()
sage: (bp_0, cp_0, ap_0,
....:  bp_2, cp_2, ap_2,
....:  bp_3, cp_3, ap_3,
....:  bm_1, cm_1, am_1) = db.parent().base_ring().gens()
sage: rqw = RightQuantumWord(
....:     q^3*bp_2*bp_0*ap_0 + q*ap_3*bm_1*am_1*bp_0)
sage: rqw.eps(3)
-q^-1 + 2*q - q^5
sage: rqw.eps(2)
-1 + 2*q - q^2 + q^3 - q^4
```

```
>>> from sage.all import *
>>> from sage.groups.braid import RightQuantumWord
>>> B = BraidGroup(Integer(3))
>>> b = B([Integer(1),-Integer(2),Integer(1),Integer(2)])
>>> db = b.deformed_burau_matrix()
>>> q = db.parent().base_ring().base_ring().gen()
>>> (bp_0, cp_0, ap_0,
...  bp_2, cp_2, ap_2,
...  bp_3, cp_3, ap_3,
...  bm_1, cm_1, am_1) = db.parent().base_ring().gens()
>>> rqw = RightQuantumWord(
...     q**Integer(3)*bp_2*bp_0*ap_0 + q*ap_3*bm_1*am_1*bp_0)
>>> rqw.eps(Integer(3))
```

(continues on next page)

(continued from previous page)

```
-q^-1 + 2*q - q^5
>>> rqw.eps(Integer(2))
-1 + 2*q - q^2 + q^3 - q^4
```

Todo

Parallelize this function, calculating all summands in the sum in parallel.

reduced_word()

Return the (reduced) right quantum word.

OUTPUT: an element in the free algebra

EXAMPLES:

```
sage: from sage.groups.braid import RightQuantumWord
sage: fig_8 = BraidGroup(3)([-1, 2, -1, 2])
sage: (
....:  bp_1, cp_1, ap_1,
....:  bp_3, cp_3, ap_3,
....:  bm_0, cm_0, am_0,
....:  bm_2, cm_2, am_2
....: ) = fig_8.deformed_burau_matrix().parent().base_ring().gens()
sage: q = bp_1.base_ring().gen()
sage: qw = RightQuantumWord(ap_1*cp_1 +
....:                           q**3*bm_2*bp_1*am_0*cm_0)
sage: qw.reduced_word()
q*cp_1*ap_1 + q^2*bp_1*cm_0*am_0*bm_2
```

```
>>> from sage.all import *
>>> from sage.groups.braid import RightQuantumWord
>>> fig_8 = BraidGroup(Integer(3))([-Integer(1), Integer(2), -Integer(1),␣
↪Integer(2)])
>>> (
...  bp_1, cp_1, ap_1,
...  bp_3, cp_3, ap_3,
...  bm_0, cm_0, am_0,
...  bm_2, cm_2, am_2
... ) = fig_8.deformed_burau_matrix().parent().base_ring().gens()
>>> q = bp_1.base_ring().gen()
>>> qw = RightQuantumWord(ap_1*cp_1 +
...                           q**Integer(3)*bm_2*bp_1*am_0*cm_0)
>>> qw.reduced_word()
q*cp_1*ap_1 + q^2*bp_1*cm_0*am_0*bm_2
```

Todo

Parallelize this function, calculating all summands in the sum in parallel.

tuples()

Get a representation of the right quantum word as a `dict`, with keys monomials in the free algebra represented as tuples and values in elements the Laurent polynomial ring in one variable.

This is in the reduced form as outlined in Definition 4.1 of [HL2018].

OUTPUT: a dict of tuples of ints corresponding to the exponents in the generators with values in the algebra's base ring

EXAMPLES:

```
sage: from sage.groups.braid import RightQuantumWord
sage: fig_8 = BraidGroup(3)([-1, 2, -1, 2])
sage: (
....:  bp_1, cp_1, ap_1,
....:  bp_3, cp_3, ap_3,
....:  bm_0, cm_0, am_0,
....:  bm_2, cm_2, am_2
....: ) = fig_8.deformed_burau_matrix().parent().base_ring().gens()
sage: q = bp_1.base_ring().gen()
sage: qw = RightQuantumWord(ap_1*cp_1 +
....:                       q**3*bm_2*bp_1*am_0*cm_0)
sage: for key, value in qw.tuples.items():
....:     print(key, value)
....:
(0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0) q
(1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0) q^2
```

```
>>> from sage.all import *
>>> from sage.groups.braid import RightQuantumWord
>>> fig_8 = BraidGroup(Integer(3))([-Integer(1), Integer(2), -Integer(1),␣
↪Integer(2)])
>>> (
...  bp_1, cp_1, ap_1,
...  bp_3, cp_3, ap_3,
...  bm_0, cm_0, am_0,
...  bm_2, cm_2, am_2
... ) = fig_8.deformed_burau_matrix().parent().base_ring().gens()
>>> q = bp_1.base_ring().gen()
>>> qw = RightQuantumWord(ap_1*cp_1 +
...                       q**Integer(3)*bm_2*bp_1*am_0*cm_0)
>>> for key, value in qw.tuples.items():
...     print(key, value)
....:
(0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0) q
(1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0) q^2
```

CHAPTER

THIRTEEN

CUBIC BRAID GROUPS

This module is devoted to factor groups of the Artin braid groups, such that the images s_i of the braid generators have order three:

$$s_i^3 = 1.$$

In general these groups have firstly been investigated by Coxeter, H.S.M. in: "Factor groups of the braid groups, Proceedings of the Fourth Canadian Mathematical Congress (Vancouver 1957), pp. 95-122".

Coxeter showed, that these groups are finite as long as the number of strands is less than 6 and infinite else-wise. More explicitly the factor group on three strand braids is isomorphic to $SL(2, 3)$, on four strand braids to $GU(3, 2)$ and on five strand braids to $Sp(4, 3) \times C_3$. Today, these finite groups are known as irreducible complex reflection groups enumerated in the Shephard-Todd classification as G_4, G_{25} and G_{32}.

Coxeter realized these groups as subgroups of unitary groups with respect to a certain Hermitian form over the complex numbers (in fact over **Q** adjoined with a primitive 12-th root of unity).

In "Einige endliche Faktorgruppen der Zopfgruppen" (Math. Z., 163 (1978), 291-302) J. Assion considered two series $S(m)$ and $U(m)$ of finite factors of these groups. The additional relations on the braid group generators $\{s_1, \cdots, s_{m-1}\}$ are

S:	$s_3 s_1 t_2 s_1 t_2^{-1} t_3 t_2 s_1 t_2^{-1} t_3^{-1} = 1$	for $m >= 5$ in case $S(m)$
U:	$t_1 t_3 = 1$	for $m >= 5$ in case $U(m)$

where $t_i = (s_i s_{i+1})^3$. He showed that each series of finite cubic braid group factors must be an epimorphic image of one of his two series, as long as the groups with less than 5 strands are the full cubic braid groups, whereas the group on 5 strands is not. He realized the groups $S(m)$ as symplectic groups over $GF(3)$ (resp. subgroups therein) and $U(m)$ as general unitary groups over $GF(4)$ (resp. subgroups therein).

All the groups considered by Coxeter and Assion are considered as finitely presented groups together with the classical realizations. It also allows for the conversion maps between the two realizations. In addition, we can construct other realizations and maps to matrix groups with help of the Burau representation. In case `gap3` and `CHEVIE` are installed, the reflection groups (via the `gap3` interface) are available, too. This can be done using the methods `as_classical_group()`, `as_matrix_group()`, `as_permutation_group()`, and `as_reflection_group()`.

REFERENCES:

- [Cox1957]
- [Ass1978]

AUTHORS:

- Sebastian Oehms 2019-02-16, initial version.

sage.groups.cubic_braid.**AssionGroupS**(*n=None*, *names='s'*)

Construct cubic braid groups `CubicBraidGroup` which have been investigated by J.Assion using the notation S(m). This function is a short hand cut for setting the construction arguments `cbg_type=CubicBraidGroup.type.AssionS` and default `names='s'`.

INPUT:

- `n` – integer (optional); the number of strands
- `names` – (default: `'s'`) string or list/tuple/iterable of strings

See also

`CubicBraidGroup`

EXAMPLES:

```
sage: S3 = AssionGroupS(3);  S3
Assion group on 3 strands of type S
sage: S3x = CubicBraidGroup(3, names='s', cbg_type=CubicBraidGroup.type.AssionS);␣
↪S3x
Assion group on 3 strands of type S
sage: S3 == S3x
True
```

```
>>> from sage.all import *
>>> S3 = AssionGroupS(Integer(3));  S3
Assion group on 3 strands of type S
>>> S3x = CubicBraidGroup(Integer(3), names='s', cbg_type=CubicBraidGroup.type.
↪AssionS); S3x
Assion group on 3 strands of type S
>>> S3 == S3x
True
```

sage.groups.cubic_braid.**AssionGroupU**(*n=None*, *names='u'*)

Construct cubic braid groups as instance of `CubicBraidGroup` which have been investigated by J.Assion using the notation U(m). This function is a short hand cut for setting the construction arguments `cbg_type=CubicBraidGroup.type.AssionU` and default `names='u'`.

INPUT:

- `n` – integer (optional); the number of strands
- `names` – (default: `'s'`) string or list/tuple/iterable of strings

See also

`CubicBraidGroup`

EXAMPLES:

```
sage: U3 = AssionGroupU(3);  U3
Assion group on 3 strands of type U
sage: U3x = CubicBraidGroup(3, names='u', cbg_type=CubicBraidGroup.type.AssionU);␣
```

(continues on next page)

(continued from previous page)

```
↪U3x
Assion group on 3 strands of type U
sage: U3 == U3x
True
```

```
>>> from sage.all import *
>>> U3 = AssionGroupU(Integer(3));  U3
Assion group on 3 strands of type U
>>> U3x = CubicBraidGroup(Integer(3), names='u', cbg_type=CubicBraidGroup.type.
↪AssionU); U3x
Assion group on 3 strands of type U
>>> U3 == U3x
True
```

class sage.groups.cubic_braid.**CubicBraidElement**(*parent*, *x*, *check=True*)

Bases: *FinitelyPresentedGroupElement*

Elements of cubic factor groups of the braid group.

For more information see *CubicBraidGroup*.

EXAMPLES:

```
sage: C4.<c1, c2, c3> = CubicBraidGroup(4); C4
Cubic Braid group on 4 strands
sage: ele1 = c1*c2*c3^-1*c2^-1
sage: ele2 = C4((1, 2, -3, -2))
sage: ele1 == ele2
True
```

```
>>> from sage.all import *
>>> C4 = CubicBraidGroup(Integer(4), names=('c1', 'c2', 'c3',)); (c1, c2, c3,) =␣
↪C4._first_ngens(3); C4
Cubic Braid group on 4 strands
>>> ele1 = c1*c2*c3**-Integer(1)*c2**-Integer(1)
>>> ele2 = C4((Integer(1), Integer(2), -Integer(3), -Integer(2)))
>>> ele1 == ele2
True
```

braid()

Return the canonical braid preimage of `self` as a `Braid`.

EXAMPLES:

```
sage: C3.<c1, c2> = CubicBraidGroup(3)
sage: c1.parent()
Cubic Braid group on 3 strands
sage: c1.braid().parent()
Braid group on 3 strands
```

```
>>> from sage.all import *
>>> C3 = CubicBraidGroup(Integer(3), names=('c1', 'c2',)); (c1, c2,) = C3._
↪first_ngens(2)
```

(continues on next page)

(continued from previous page)

```
>>> c1.parent()
Cubic Braid group on 3 strands
>>> c1.braid().parent()
Braid group on 3 strands
```

burau_matrix(*root_bur=None*, *domain=None*, *characteristic=None*, *var='t'*, *reduced=False*)

Return the Burau matrix of the cubic braid coset.

This method uses the same method belonging to `Braid`, but reduces the indeterminate to a primitive sixth (resp. twelfth in case `reduced='unitary'`) root of unity.

INPUT (all arguments are optional keywords):

- `root_bur` – six (resp. twelfth) root of unity in some field (default: root of unity over **Q**)
- `domain` – (default: cyclotomic field of order 3 and degree 2, resp. the domain of `root_bur` if given) base ring for the Burau matrix
- `characteristic` – integer giving the characteristic of the domain (default: 0 or the characteristic of `domain` if given)
- `var` – string used for the indeterminate name in case `root_bur` must be constructed in a splitting field
- `reduced` – boolean or string (default: `False`); for more information see the documentation of *burau_matrix()* of `Braid`

OUTPUT:

The Burau matrix of the cubic braid coset with entries in the domain given by the options. In case the option `reduced='unitary'` is given a triple consisting of the Burau matrix, its adjoined and the Hermitian form is returned.

EXAMPLES:

```
sage: C3.<c1, c2> = CubicBraidGroup(3)
sage: ele = c1*c2*c1

sage: # needs sage.rings.number_field
sage: BuMa = ele.burau_matrix(); BuMa
[  -zeta3          1      zeta3]
[  -zeta3 zeta3 + 1          0]
[       1          0          0]
sage: BuMa.base_ring()
Cyclotomic Field of order 3 and degree 2
sage: BuMa == ele.burau_matrix(characteristic=0)
True
sage: BuMa = ele.burau_matrix(domain=QQ); BuMa
[-t + 1      1  t - 1]
[-t + 1      t      0]
[     1      0      0]
sage: BuMa.base_ring()
Number Field in t with defining polynomial t^2 - t + 1
sage: BuMa = ele.burau_matrix(domain = QQ[I, sqrt(3)]); BuMa              #␣
↪needs sage.symbolic
[ 1/2*sqrt3*I + 1/2                  1 -1/2*sqrt3*I - 1/2]
[ 1/2*sqrt3*I + 1/2 -1/2*sqrt3*I + 1/2                  0]
[                 1                  0                  0]
```

(continues on next page)

(continued from previous page)

```
sage: BuMa.base_ring()                                                      #␣
↪needs sage.symbolic
Number Field in I with defining polynomial x^2 + 1 over its base field

sage: BuMa = ele.burau_matrix(characteristic=7); BuMa
[3 1 4]
[3 5 0]
[1 0 0]
sage: BuMa.base_ring()
Finite Field of size 7

sage: # needs sage.rings.finite_rings
sage: BuMa = ele.burau_matrix(characteristic=2); BuMa
[t + 1     1 t + 1]
[t + 1     t     0]
[    1     0     0]
sage: BuMa.base_ring()
Finite Field in t of size 2^2
sage: F4.<r64> = GF(4)
sage: BuMa = ele.burau_matrix(root_bur=r64); BuMa
[r64 + 1       1 r64 + 1]
[r64 + 1     r64       0]
[      1       0       0]
sage: BuMa.base_ring()
Finite Field in r64 of size 2^2
sage: BuMa = ele.burau_matrix(domain=GF(5)); BuMa
[2*t + 2       1 3*t + 3]
[2*t + 2 3*t + 4       0]
[      1       0       0]
sage: BuMa.base_ring()
Finite Field in t of size 5^2

sage: # needs sage.rings.number_field
sage: BuMa, BuMaAd, H = ele.burau_matrix(reduced='unitary'); BuMa
[       0 zeta12^3]
[zeta12^3        0]
sage: BuMa * H * BuMaAd == H
True
sage: BuMa.base_ring()
Cyclotomic Field of order 12 and degree 4
sage: BuMa, BuMaAd, H = ele.burau_matrix(domain=QQ[I, sqrt(3)],             #␣
↪needs sage.symbolic
....:                                    reduced='unitary'); BuMa
[0 I]
[I 0]
sage: BuMa.base_ring()                                                      #␣
↪needs sage.symbolic
Number Field in I with defining polynomial x^2 + 1 over its base field
```

```
>>> from sage.all import *
>>> C3 = CubicBraidGroup(Integer(3), names=('c1', 'c2',)); (c1, c2,) = C3._
↪first_ngens(2)
```

(continues on next page)

(continued from previous page)

```
>>> ele = c1*c2*c1

>>> # needs sage.rings.number_field
>>> BuMa = ele.burau_matrix(); BuMa
[  -zeta3          1      zeta3]
[  -zeta3  zeta3 + 1          0]
[       1          0          0]
>>> BuMa.base_ring()
Cyclotomic Field of order 3 and degree 2
>>> BuMa == ele.burau_matrix(characteristic=Integer(0))
True
>>> BuMa = ele.burau_matrix(domain=QQ); BuMa
[-t + 1      1  t - 1]
[-t + 1      t      0]
[     1      0      0]
>>> BuMa.base_ring()
Number Field in t with defining polynomial t^2 - t + 1
>>> BuMa = ele.burau_matrix(domain = QQ[I, sqrt(Integer(3))]); BuMa
↪      # needs sage.symbolic
[ 1/2*sqrt3*I + 1/2                  1 -1/2*sqrt3*I - 1/2]
[ 1/2*sqrt3*I + 1/2 -1/2*sqrt3*I + 1/2                  0]
[                 1                  0                  0]
>>> BuMa.base_ring()                                                    #
↪needs sage.symbolic
Number Field in I with defining polynomial x^2 + 1 over its base field

>>> BuMa = ele.burau_matrix(characteristic=Integer(7)); BuMa
[3 1 4]
[3 5 0]
[1 0 0]
>>> BuMa.base_ring()
Finite Field of size 7

>>> # needs sage.rings.finite_rings
>>> BuMa = ele.burau_matrix(characteristic=Integer(2)); BuMa
[t + 1     1 t + 1]
[t + 1     t     0]
[    1     0     0]
>>> BuMa.base_ring()
Finite Field in t of size 2^2
>>> F4 = GF(Integer(4), names=('r64',)); (r64,) = F4._first_ngens(1)
>>> BuMa = ele.burau_matrix(root_bur=r64); BuMa
[r64 + 1       1 r64 + 1]
[r64 + 1     r64       0]
[      1       0       0]
>>> BuMa.base_ring()
Finite Field in r64 of size 2^2
>>> BuMa = ele.burau_matrix(domain=GF(Integer(5))); BuMa
[2*t + 2       1 3*t + 3]
[2*t + 2 3*t + 4       0]
[      1       0       0]
>>> BuMa.base_ring()
```

(continues on next page)

(continued from previous page)

```
Finite Field in t of size 5^2

>>> # needs sage.rings.number_field
>>> BuMa, BuMaAd, H = ele.burau_matrix(reduced='unitary'); BuMa
[       0 zeta12^3]
[zeta12^3        0]
>>> BuMa * H * BuMaAd == H
True
>>> BuMa.base_ring()
Cyclotomic Field of order 12 and degree 4
>>> BuMa, BuMaAd, H = ele.burau_matrix(domain=QQ[I, sqrt(Integer(3))],       ␣
↪     # needs sage.symbolic
...                                       reduced='unitary'); BuMa
[0 I]
[I 0]
>>> BuMa.base_ring()                                                        #␣
↪needs sage.symbolic
Number Field in I with defining polynomial x^2 + 1 over its base field
```

class sage.groups.cubic_braid.**CubicBraidGroup**(*names*, *cbg_type=None*)

Bases: UniqueRepresentation, *FinitelyPresentedGroup*

Factor groups of the Artin braid group mapping their generators to elements of order 3.

These groups are implemented as a particular case of finitely presented groups similar to the BraidGroup_class.

A cubic braid group can be created by giving the number of strands, and the name of the generators in a similar way as it works for the BraidGroup_class.

INPUT:

- names – see the corresponding documentation of BraidGroup_class
- cbg_type – (default: CubicBraidGroup.type.Coxeter; see explanation below) enum type *CubicBraidGroup.type*

Setting the keyword cbg_type to one on the values CubicBraidGroup.type.AssionS or CubicBraidGroup.type.AssionU, the additional relations due to Assion are added:

$$\begin{array}{lll} \text{S:} & s_3 s_1 t_2 s_1 t_2^{-1} t_3 t_2 s_1 t_2^{-1} t_3^{-1} = 1 & \text{for } m >= 5 \text{ in case } S(m), \\ \text{U:} & t_1 t_3 = 1 & \text{for } m >= 5 \text{ in case } U(m), \end{array}$$

where $t_i = (s_i s_{i+1})^3$. If cbg_type == CubicBraidGroup.type.Coxeter (default) only the cubic relation on the generators is active (Coxeter's case of investigation). Note that for $n = 2, 3, 4$, the groups do not differ between the three possible values of cbg_type (as finitely presented groups). However, the CubicBraidGroup.type.Coxeter, CubicBraidGroup.type.AssionS and CubicBraidGroup.type.AssionU are different, so they have different classical realizations implemented.

See also

Instances can also be constructed more easily by using *CubicBraidGroup()*, *AssionGroupS()* and *AssionGroupU()*.

EXAMPLES:

```
sage: U3 = CubicBraidGroup(3, cbg_type=CubicBraidGroup.type.AssionU); U3
Assion group on 3 strands of type U
sage: U3.gens()
(c0, c1)
```

```
>>> from sage.all import *
>>> U3 = CubicBraidGroup(Integer(3), cbg_type=CubicBraidGroup.type.AssionU); U3
Assion group on 3 strands of type U
>>> U3.gens()
(c0, c1)
```

Alternative possibilities defining `U3`:

```
sage: U3 = AssionGroupU(3); U3
Assion group on 3 strands of type U
sage: U3.gens()
(u0, u1)
sage: U3.<u1,u2> = AssionGroupU(3); U3
Assion group on 3 strands of type U
sage: U3.gens()
(u1, u2)
```

```
>>> from sage.all import *
>>> U3 = AssionGroupU(Integer(3)); U3
Assion group on 3 strands of type U
>>> U3.gens()
(u0, u1)
>>> U3 = AssionGroupU(Integer(3), names=('u1', 'u2',)); (u1, u2,) = U3._first_
↪ngens(2); U3
Assion group on 3 strands of type U
>>> U3.gens()
(u1, u2)
```

Alternates naming the generators:

```
sage: U3 = AssionGroupU(3, 'a, b'); U3
Assion group on 3 strands of type U
sage: U3.gens()
(a, b)
sage: C3 = CubicBraidGroup(3, 't'); C3
Cubic Braid group on 3 strands
sage: C3.gens()
(t0, t1)
sage: U3.is_isomorphic(C3)
True
sage: U3.as_classical_group()
Subgroup generated by [(1,7,6)(3,19,14)(4,15,10)(5,11,18)(12,16,20),
                        (1,12,13)(2,15,19)(4,9,14)(5,18,8)(6,21,16)]
 of (The projective general unitary group of degree 3 over Finite Field of size 2)
sage: C3.as_classical_group()
Subgroup with 2 generators (
[  E(3)^2        0]  [        1 -E(12)^7]
[-E(12)^7        1], [        0   E(3)^2]
```

(continues on next page)

(continued from previous page)

```
) of General Unitary Group of degree 2 over Universal Cyclotomic Field
with respect to positive definite hermitian form
[-E(12)^7 + E(12)^11                    -1]
[                  -1 -E(12)^7 + E(12)^11]
```

```
>>> from sage.all import *
>>> U3 = AssionGroupU(Integer(3), 'a, b'); U3
Assion group on 3 strands of type U
>>> U3.gens()
(a, b)
>>> C3 = CubicBraidGroup(Integer(3), 't'); C3
Cubic Braid group on 3 strands
>>> C3.gens()
(t0, t1)
>>> U3.is_isomorphic(C3)
True
>>> U3.as_classical_group()
Subgroup generated by [(1,7,6)(3,19,14)(4,15,10)(5,11,18)(12,16,20),
                        (1,12,13)(2,15,19)(4,9,14)(5,18,8)(6,21,16)]
 of (The projective general unitary group of degree 3 over Finite Field of size 2)
>>> C3.as_classical_group()
Subgroup with 2 generators (
[  E(3)^2        0]  [        1 -E(12)^7]
[-E(12)^7        1], [        0   E(3)^2]
) of General Unitary Group of degree 2 over Universal Cyclotomic Field
with respect to positive definite hermitian form
[-E(12)^7 + E(12)^11                    -1]
[                  -1 -E(12)^7 + E(12)^11]
```

REFERENCES:

- [Cox1957]
- [Ass1978]

Element

alias of *CubicBraidElement*

as_classical_group(*embedded=False*)

Create an isomorphic image of `self` as a classical group according to the construction given by Coxeter resp. Assion.

INPUT:

- `embedded` – boolean (default: `False`); this boolean effects the cases of Assion groups when they are realized as projective groups only. More precisely: if `self` is of `cbg_type` `CubicBraidGroup.type.AssionS` (for example) and the number of strands `n` is even, than its classical group is a subgroup of `PSp(n,3)` (being centralized by the element `self.centralizing_element(projective=True))`. By default this group will be given. Setting `embedded = True` the classical realization is given as subgroup of its classical enlargement with one more strand (in this case as subgroup of `Sp(n,3))`.

OUTPUT:

Depending on the type of `self` and the number of strands an instance of `Sp(n-1,3)`, `GU(n-1,2)`, subgroup of `PSp(n,3)`, `PGU(n,2)`, or a subgroup of `GU(n-1, UCF)` (`cbg_type == CubicBraidGroup.type.`

`Coxeter`) with respect to a certain Hermitian form attached to the Burau representation (used by Coxeter and Squier). Here `UCF` stands for the universal cyclotomic field.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: U3 = AssionGroupU(3)
sage: U3Cl = U3.as_classical_group(); U3Cl
Subgroup generated by [(1,7,6)(3,19,14)(4,15,10)(5,11,18)(12,16,20),
                       (1,12,13)(2,15,19)(4,9,14)(5,18,8)(6,21,16)] of
 (The projective general unitary group of degree 3 over Finite Field of size␣
↪2)
sage: U3Clemb = U3.as_classical_group(embedded=True); U3Clemb
Subgroup with 2 generators (
[0 0 a]  [a + 1     a     a]
[0 1 0]  [    a a + 1     a]
[a 0 a], [    a     a a + 1]
) of General Unitary Group of degree 3 over Finite Field in a of size 2^2
sage: u = U3([-2,1,-2,1]); u
(u1^-1*u0)^2
sage: uCl = U3Cl(u); uCl
(1,16)(2,9)(3,10)(4,19)(6,12)(7,20)(13,21)(14,15)
sage: uCle = U3Clemb(u); uCle
[a + 1 a + 1     1]
[a + 1     0     a]
[    1     a     a]
sage: U3(uCl) == u
True
sage: U3(uCle) == u
True
sage: U4 = AssionGroupU(4)
sage: U4Cl = U4.as_classical_group(); U4Cl
General Unitary Group of degree 3 over Finite Field in a of size 2^2
sage: U3Clemb.ambient() == U4Cl
True

sage: # needs sage.rings.number_field
sage: C4 = CubicBraidGroup(4)
sage: C4Cl = C4.as_classical_group(); C4Cl
Subgroup with 3 generators (
[  E(3)^2        0        0]  [        1 -E(12)^7        0]
[-E(12)^7        1        0]  [        0   E(3)^2        0]
[       0        0        1], [        0 -E(12)^7        1],

[       1        0        0]
[       0        1 -E(12)^7]
[       0        0   E(3)^2]
) of General Unitary Group of degree 3 over Universal Cyclotomic Field
with respect to positive definite hermitian form
[-E(12)^7 + E(12)^11                  -1                   0]
[                 -1 -E(12)^7 + E(12)^11                  -1]
[                  0                  -1 -E(12)^7 + E(12)^11]
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> U3 = AssionGroupU(Integer(3))
>>> U3Cl = U3.as_classical_group(); U3Cl
Subgroup generated by [(1,7,6)(3,19,14)(4,15,10)(5,11,18)(12,16,20),
                       (1,12,13)(2,15,19)(4,9,14)(5,18,8)(6,21,16)] of
 (The projective general unitary group of degree 3 over Finite Field of size␣
↪2)
>>> U3Clemb = U3.as_classical_group(embedded=True); U3Clemb
Subgroup with 2 generators (
[0 0 a]  [a + 1     a     a]
[0 1 0]  [    a a + 1     a]
[a 0 a], [    a     a a + 1]
) of General Unitary Group of degree 3 over Finite Field in a of size 2^2
>>> u = U3([-Integer(2),Integer(1),-Integer(2),Integer(1)]); u
(u1^-1*u0)^2
>>> uCl = U3Cl(u); uCl
(1,16)(2,9)(3,10)(4,19)(6,12)(7,20)(13,21)(14,15)
>>> uCle = U3Clemb(u); uCle
[a + 1 a + 1     1]
[a + 1     0     a]
[    1     a     a]
>>> U3(uCl) == u
True
>>> U3(uCle) == u
True
>>> U4 = AssionGroupU(Integer(4))
>>> U4Cl = U4.as_classical_group(); U4Cl
General Unitary Group of degree 3 over Finite Field in a of size 2^2
>>> U3Clemb.ambient() == U4Cl
True

>>> # needs sage.rings.number_field
>>> C4 = CubicBraidGroup(Integer(4))
>>> C4Cl = C4.as_classical_group(); C4Cl
Subgroup with 3 generators (
[  E(3)^2         0         0]  [        1 -E(12)^7         0]
[-E(12)^7         1         0]  [        0   E(3)^2         0]
[        0        0         1], [        0 -E(12)^7         1],
<BLANKLINE>
[        1        0         0]
[        0        1 -E(12)^7]
[        0        0   E(3)^2]
) of General Unitary Group of degree 3 over Universal Cyclotomic Field
with respect to positive definite hermitian form
[-E(12)^7 + E(12)^11                   -1                    0]
[                  -1 -E(12)^7 + E(12)^11                   -1]
[                   0                   -1 -E(12)^7 + E(12)^11]
```

as_matrix_group(*root_bur=None*, *domain=None*, *characteristic=None*, *var='t'*, *reduced=False*)

Create an epimorphic image of `self` as a matrix group by use of the burau representation.

INPUT:

- `root_bur` – (default: root of unity over **Q**) six (resp. twelfth) root of unity in some field
- `domain` – (default: cyclotomic field of order 3 and degree 2, resp. the domain of `root_bur` if given) base ring for the Burau matrix
- `characteristic` – integer (optional); the characteristic of the domain; if none of the keywords `root_bur`, `domain` and `characteristic` are given, the default characteristic is 3 (resp. 2) if `self` is of `cbg_type` `CubicBraidGroup.type.AssionS` (resp. `CubicBraidGroup.type.AssionU`)
- `var` – string used for the indeterminate name in case `root_bur` must be constructed in a splitting field
- `reduced` – boolean (default: `False`); for more information see the documentation of `Braid.burau_matrix()`

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: C5 = CubicBraidGroup(5)
sage: C5Mch5 = C5.as_matrix_group(characteristic=5); C5Mch5
Matrix group over Finite Field in t of size 5^2 with 4 generators (
[2*t + 2 3*t + 4       0       0       0]
[      1       0       0       0       0]
[      0       0       1       0       0]
[      0       0       0       1       0]
[      0       0       0       0       1],

[      1       0       0       0       0]
[      0 2*t + 2 3*t + 4       0       0]
[      0       1       0       0       0]
[      0       0       0       1       0]
[      0       0       0       0       1],

[      1       0       0       0       0]
[      0       1       0       0       0]
[      0       0 2*t + 2 3*t + 4       0]
[      0       0       1       0       0]
[      0       0       0       0       1],

[      1       0       0       0       0]
[      0       1       0       0       0]
[      0       0       1       0       0]
[      0       0       0 2*t + 2 3*t + 4]
[      0       0       0       1       0]
)
sage: c = C5([3,4,-2,-3,1]); c
c2*c3*c1^-1*c2^-1*c0
sage: m = C5Mch5(c); m
[2*t + 2 3*t + 4       0       0       0]
[      0       0       0       1       0]
[2*t + 1       0 2*t + 2     3*t 3*t + 3]
[2*t + 2       0       0 3*t + 4       0]
[      0       0 2*t + 2 3*t + 4       0]
sage: m_back = C5(m)
sage: m_back == c
True
sage: U5 = AssionGroupU(5); U5
```

(continues on next page)

(continued from previous page)

```
Assion group on 5 strands of type U
sage: U5Mch3 = U5.as_matrix_group(characteristic=3)
Traceback (most recent call last):
...
ValueError: Burau representation does not factor through the relations
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> C5 = CubicBraidGroup(Integer(5))
>>> C5Mch5 = C5.as_matrix_group(characteristic=Integer(5)); C5Mch5
Matrix group over Finite Field in t of size 5^2 with 4 generators (
[2*t + 2 3*t + 4       0       0       0]
[      1       0       0       0       0]
[      0       0       1       0       0]
[      0       0       0       1       0]
[      0       0       0       0       1],
<BLANKLINE>
[      1       0       0       0       0]
[      0 2*t + 2 3*t + 4       0       0]
[      0       1       0       0       0]
[      0       0       0       1       0]
[      0       0       0       0       1],
<BLANKLINE>
[      1       0       0       0       0]
[      0       1       0       0       0]
[      0       0 2*t + 2 3*t + 4       0]
[      0       0       1       0       0]
[      0       0       0       0       1],
<BLANKLINE>
[      1       0       0       0       0]
[      0       1       0       0       0]
[      0       0       1       0       0]
[      0       0       0 2*t + 2 3*t + 4]
[      0       0       0       1       0]
)
>>> c = C5([Integer(3),Integer(4),-Integer(2),-Integer(3),Integer(1)]); c
c2*c3*c1^-1*c2^-1*c0
>>> m = C5Mch5(c); m
[2*t + 2 3*t + 4       0       0       0]
[      0       0       0       1       0]
[2*t + 1       0 2*t + 2     3*t 3*t + 3]
[2*t + 2       0       0 3*t + 4       0]
[      0       0 2*t + 2 3*t + 4       0]
>>> m_back = C5(m)
>>> m_back == c
True
>>> U5 = AssionGroupU(Integer(5)); U5
Assion group on 5 strands of type U
>>> U5Mch3 = U5.as_matrix_group(characteristic=Integer(3))
Traceback (most recent call last):
...
ValueError: Burau representation does not factor through the relations
```

as_permutation_group(*use_classical=True*)

Return a permutation group isomorphic to `self` that has a group isomorphism from `self` as a conversion.

INPUT:

- `use_classical` – boolean (default: `True`); the permutation group is calculated via the attached classical matrix group as this results in a smaller degree; if `False`, the permutation group will be calculated using `self` (as finitely presented group)

EXAMPLES:

```
sage: C3 = CubicBraidGroup(3)
sage: PC3 = C3.as_permutation_group()
sage: assert C3.is_isomorphic(PC3)   # random (with respect to the occurrence
↪of the info message)
sage: PC3.degree()
8
sage: c = C3([2,1-2])
sage: C3(PC3(c)) == c
True
```

```
>>> from sage.all import *
>>> C3 = CubicBraidGroup(Integer(3))
>>> PC3 = C3.as_permutation_group()
>>> assert C3.is_isomorphic(PC3)   # random (with respect to the occurrence of
↪the info message)
>>> PC3.degree()
8
>>> c = C3([Integer(2),Integer(1)-Integer(2)])
>>> C3(PC3(c)) == c
True
```

as_reflection_group()

Return an isomorphic image of `self` as irreducible complex reflection group.

This is possible only for the finite cubic braid groups of `cbg_type` `CubicBraidGroup.type.Coxeter`.

> **Note**
>
> This method uses the sage implementation of reflection group via the `gap3` `CHEVIE` package. These must be installed in order to use this method.

EXAMPLES:

```
sage: # optional - gap3
sage: C3.<c1,c2> = CubicBraidGroup(3)
sage: R3 = C3.as_reflection_group(); R3
Irreducible complex reflection group of rank 2 and type ST4
sage: R3.cartan_matrix()
[-2*E(3) - E(3)^2           E(3)^2]
[        -E(3)^2 -2*E(3) - E(3)^2]
sage: R3.simple_roots()
Finite family {1: (0, -2*E(3) - E(3)^2), 2: (2*E(3)^2, E(3)^2)}
```

(continues on next page)

(continued from previous page)

```
sage: R3.simple_coroots()
Finite family {1: (0, 1), 2: (1/3*E(3) - 1/3*E(3)^2, 1/3*E(3) - 1/3*E(3)^2)}
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> C3 = CubicBraidGroup(Integer(3), names=('c1', 'c2',)); (c1, c2,) = C3._
↪first_ngens(2)
>>> R3 = C3.as_reflection_group(); R3
Irreducible complex reflection group of rank 2 and type ST4
>>> R3.cartan_matrix()
[-2*E(3) - E(3)^2           E(3)^2]
[         -E(3)^2 -2*E(3) - E(3)^2]
>>> R3.simple_roots()
Finite family {1: (0, -2*E(3) - E(3)^2), 2: (2*E(3)^2, E(3)^2)}
>>> R3.simple_coroots()
Finite family {1: (0, 1), 2: (1/3*E(3) - 1/3*E(3)^2, 1/3*E(3) - 1/3*E(3)^2)}
```

Conversion maps:

```
sage: # optional - gap3
sage: r = R3.an_element()
sage: cr = C3(r); cr
c1*c2
sage: mr = r.matrix(); mr
[ 1/3*E(3) - 1/3*E(3)^2  2/3*E(3) + 1/3*E(3)^2]
[-2/3*E(3) + 2/3*E(3)^2  2/3*E(3) + 1/3*E(3)^2]
sage: C3Cl = C3.as_classical_group()
sage: C3Cl(cr)
[ E(3)^2    -E(4)]
[-E(12)^7        0]
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> r = R3.an_element()
>>> cr = C3(r); cr
c1*c2
>>> mr = r.matrix(); mr
[ 1/3*E(3) - 1/3*E(3)^2  2/3*E(3) + 1/3*E(3)^2]
[-2/3*E(3) + 2/3*E(3)^2  2/3*E(3) + 1/3*E(3)^2]
>>> C3Cl = C3.as_classical_group()
>>> C3Cl(cr)
[ E(3)^2    -E(4)]
[-E(12)^7        0]
```

The reflection groups can also be viewed as subgroups of unitary groups over the universal cyclotomic field. Note that the unitary group corresponding to the reflection group is isomorphic but different from the classical group due to different hermitian forms for the unitary groups they live in:

```
sage: # optional - gap3
sage: C4 = CubicBraidGroup(4)
sage: R4 = C4.as_reflection_group()
sage: R4.invariant_form()
```

(continues on next page)

(continued from previous page)

```
[1 0 0]
[0 1 0]
[0 0 1]
sage: _ == C4.classical_invariant_form()
False
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> C4 = CubicBraidGroup(Integer(4))
>>> R4 = C4.as_reflection_group()
>>> R4.invariant_form()
[1 0 0]
[0 1 0]
[0 0 1]
>>> _ == C4.classical_invariant_form()
False
```

braid_group()

Return a *BraidGroup* with identical generators, such that there exists an epimorphism to `self`.

OUTPUT: a *BraidGroup* having conversion maps to and from `self` (which is just a section in the latter case)

EXAMPLES:

```
sage: U5 = AssionGroupU(5); U5
Assion group on 5 strands of type U
sage: B5 = U5.braid_group(); B5
Braid group on 5 strands
sage: b = B5([4,3,2,-4,1])
sage: u = U5([4,3,2,-4,1])
sage: u == b
False
sage: b.burau_matrix()
[ 1 - t      t      0      0      0]
[ 1 - t      0      t      0      0]
[ 1 - t      0      0      0      t]
[ 1 - t      0      0      1 -1 + t]
[     1      0      0      0      0]
sage: u.burau_matrix()
[t + 1     t     0     0     0]
[t + 1     0     t     0     0]
[t + 1     0     0     0     t]
[t + 1     0     0     1 t + 1]
[    1     0     0     0     0]
sage: bU = U5(b)
sage: uB = B5(u)
sage: bU == u
True
sage: uB == b
True
```

```
>>> from sage.all import *
>>> U5 = AssionGroupU(Integer(5)); U5
Assion group on 5 strands of type U
>>> B5 = U5.braid_group(); B5
Braid group on 5 strands
>>> b = B5([Integer(4),Integer(3),Integer(2),-Integer(4),Integer(1)])
>>> u = U5([Integer(4),Integer(3),Integer(2),-Integer(4),Integer(1)])
>>> u == b
False
>>> b.burau_matrix()
[ 1 - t      t      0      0      0]
[ 1 - t      0      t      0      0]
[ 1 - t      0      0      0      t]
[ 1 - t      0      0      1 -1 + t]
[     1      0      0      0      0]
>>> u.burau_matrix()
[t + 1     t     0     0     0]
[t + 1     0     t     0     0]
[t + 1     0     0     0     t]
[t + 1     0     0     1 t + 1]
[    1     0     0     0     0]
>>> bU = U5(b)
>>> uB = B5(u)
>>> bU == u
True
>>> uB == b
True
```

cardinality()

To avoid long wait-time on calculations the order will be obtained using the classical realization.

OUTPUT: cardinality of the group as integer or infinity

EXAMPLES:

```
sage: S15 = AssionGroupS(15)
sage: S15.order()
109777561863482259035023554842176139436811616256000
sage: C6 = CubicBraidGroup(6)
sage: C6.order()
+Infinity
```

```
>>> from sage.all import *
>>> S15 = AssionGroupS(Integer(15))
>>> S15.order()
109777561863482259035023554842176139436811616256000
>>> C6 = CubicBraidGroup(Integer(6))
>>> C6.order()
+Infinity
```

centralizing_element(*embedded=False*)

Return the centralizing element defined by the work of Assion (Hilfssatz 1.1.3 and 1.2.3).

INPUT:

- embedded – boolean (default; False); this boolean only effects the cases of Assion groups when they are realized as projective groups. More precisely: if self is of cbg_type CubicBraidGroup.type.AssionS (for example) and the number of strands n is even, than its classical group is a subgroup of PSp(n,3) being centralized by the element return for option embedded=False. Otherwise the image of this element inside the embedded classical group will be returned (see option embedded of classical_group()).

OUTPUT:

Depending on the optional keyword a permutation as an element of PSp(n,3) (type S) or PGU(n,2) (type U) for n = 0 mod 2 (type S) resp. n = 0 mod 3 (type U) is returned. Otherwise, the centralizing element is a matrix belonging to Sp(n,3) resp. GU(n,2).

EXAMPLES:

```
sage: U3 = AssionGroupU(3);  U3
Assion group on 3 strands of type U
sage: U3Cl = U3.as_classical_group(); U3Cl
Subgroup generated by [(1,7,6)(3,19,14)(4,15,10)(5,11,18)(12,16,20),
                        (1,12,13)(2,15,19)(4,9,14)(5,18,8)(6,21,16)]
 of (The projective general unitary group of degree 3 over Finite Field of␣
↪size 2)
sage: c = U3.centralizing_element(); c
(1,16)(2,9)(3,10)(4,19)(6,12)(7,20)(13,21)(14,15)
sage: c in U3Cl
True
sage: P = U3Cl.ambient_group()
sage: P.centralizer(c) == U3Cl
True
```

```
>>> from sage.all import *
>>> U3 = AssionGroupU(Integer(3));  U3
Assion group on 3 strands of type U
>>> U3Cl = U3.as_classical_group(); U3Cl
Subgroup generated by [(1,7,6)(3,19,14)(4,15,10)(5,11,18)(12,16,20),
                        (1,12,13)(2,15,19)(4,9,14)(5,18,8)(6,21,16)]
 of (The projective general unitary group of degree 3 over Finite Field of␣
↪size 2)
>>> c = U3.centralizing_element(); c
(1,16)(2,9)(3,10)(4,19)(6,12)(7,20)(13,21)(14,15)
>>> c in U3Cl
True
>>> P = U3Cl.ambient_group()
>>> P.centralizer(c) == U3Cl
True
```

Embedded version:

```
sage: cm = U3.centralizing_element(embedded=True); cm
[a + 1 a + 1     1]
[a + 1     0     a]
[    1     a     a]
sage: U4 = AssionGroupU(4)
sage: U4Cl = U4.as_classical_group()
sage: cm in U4Cl
```

(continues on next page)

(continued from previous page)

```
True
sage: [cm * U4Cl(g) == U4Cl(g) * cm for g in U4.gens()]
[True, True, False]
```

```
>>> from sage.all import *
>>> cm = U3.centralizing_element(embedded=True); cm
[a + 1 a + 1      1]
[a + 1      0      a]
[    1      a      a]
>>> U4 = AssionGroupU(Integer(4))
>>> U4Cl = U4.as_classical_group()
>>> cm in U4Cl
True
>>> [cm * U4Cl(g) == U4Cl(g) * cm for g in U4.gens()]
[True, True, False]
```

classical_invariant_form()

Return the invariant form of the classical realization of `self`.

OUTPUT:

A square matrix of dimension according to the space the classical realization is operating on. In the case of the full cubic braid groups and of the Assion groups of `cbg_type CubicBraidGroup.type.AssionU` the matrix is Hermitian. In the case of the Assion groups of `cbg_type CubicBraidGroup.type.AssionS` it is alternating. Note that the invariant form of the full cubic braid group on more than 5 strands is degenerated (causing the group to be infinite).

In the case of Assion groups having projective classical groups, the invariant form corresponds to the ambient group of its classical embedding.

EXAMPLES:

```
sage: S3 = AssionGroupS(3)
sage: S3.classical_invariant_form()
[0 1]
[2 0]
sage: S4 = AssionGroupS(4)
sage: S4.classical_invariant_form()
[0 0 0 1]
[0 0 1 0]
[0 2 0 0]
[2 0 0 0]
sage: S5 = AssionGroupS(5)
sage: S4.classical_invariant_form() == S5.classical_invariant_form()
True
sage: U4 = AssionGroupU(4)
sage: U4.classical_invariant_form()
[0 0 1]
[0 1 0]
[1 0 0]
sage: C5 = CubicBraidGroup(5)
sage: C5.classical_invariant_form()
[-E(12)^7 + E(12)^11                  -1                   0                   ␣
↪ 0]
```

(continues on next page)

(continued from previous page)

```
[                -1 -E(12)^7 + E(12)^11                  -1                  ␣
↪0]
[                 0                  -1 -E(12)^7 + E(12)^11                  -
↪1]
[                 0                   0                  -1 -E(12)^7 + E(12)^
↪11]
sage: _.is_singular()
False
sage: C6 = CubicBraidGroup(6)
sage: C6.classical_invariant_form().is_singular()
True
```

```
>>> from sage.all import *
>>> S3 = AssionGroupS(Integer(3))
>>> S3.classical_invariant_form()
[0 1]
[2 0]
>>> S4 = AssionGroupS(Integer(4))
>>> S4.classical_invariant_form()
[0 0 0 1]
[0 0 1 0]
[0 2 0 0]
[2 0 0 0]
>>> S5 = AssionGroupS(Integer(5))
>>> S4.classical_invariant_form() == S5.classical_invariant_form()
True
>>> U4 = AssionGroupU(Integer(4))
>>> U4.classical_invariant_form()
[0 0 1]
[0 1 0]
[1 0 0]
>>> C5 = CubicBraidGroup(Integer(5))
>>> C5.classical_invariant_form()
[-E(12)^7 + E(12)^11                  -1                   0                  ␣
↪ 0]
[                -1 -E(12)^7 + E(12)^11                  -1                  ␣
↪0]
[                 0                  -1 -E(12)^7 + E(12)^11                  -
↪1]
[                 0                   0                  -1 -E(12)^7 + E(12)^
↪11]
>>> _.is_singular()
False
>>> C6 = CubicBraidGroup(Integer(6))
>>> C6.classical_invariant_form().is_singular()
True
```

codegrees()

Return the codegrees of `self`.

This only makes sense when `self` is a finite reflection group.

EXAMPLES:

```
sage: CubicBraidGroup(5).codegrees()
(0, 6, 12, 18)
```

```
>>> from sage.all import *
>>> CubicBraidGroup(Integer(5)).codegrees()
(0, 6, 12, 18)
```

cubic_braid_subgroup(*nstrands=None*)

Return a cubic braid group as subgroup of `self` on the first `nstrands` strands.

INPUT:

- `nstrands` – (default: `self.strands() - 1`) integer at least 1 and at most `self.strands()` giving the number of strands of the subgroup

> **Warning**
>
> Since `self` is inherited from `UniqueRepresentation`, the obtained instance is identical to other instances created with the same arguments (see example below). The ambient group corresponds to the last call of this method.

EXAMPLES:

```
sage: U5 = AssionGroupU(5)
sage: U3s = U5.cubic_braid_subgroup(3)
sage: u1, u2 = U3s.gens()
sage: u1 in U5
False
sage: U5(u1) in U5.gens()
True
sage: U3s is AssionGroupU(3)
True
sage: U3s.ambient() == U5
True
```

```
>>> from sage.all import *
>>> U5 = AssionGroupU(Integer(5))
>>> U3s = U5.cubic_braid_subgroup(Integer(3))
>>> u1, u2 = U3s.gens()
>>> u1 in U5
False
>>> U5(u1) in U5.gens()
True
>>> U3s is AssionGroupU(Integer(3))
True
>>> U3s.ambient() == U5
True
```

degrees()

Return the degrees of `self`.

This only makes sense when `self` is a finite reflection group.

EXAMPLES:

```
sage: CubicBraidGroup(4).degrees()
(6, 9, 12)
```

```
>>> from sage.all import *
>>> CubicBraidGroup(Integer(4)).degrees()
(6, 9, 12)
```

index_set()

Return the index set of `self`.

This is the set of integers $0, \ldots, n-2$ where n is the number of strands.

This is only used when `self` is a finite reflection group.

EXAMPLES:

```
sage: CubicBraidGroup(3).index_set()
[0, 1]
```

```
>>> from sage.all import *
>>> CubicBraidGroup(Integer(3)).index_set()
[0, 1]
```

is_finite()

Return if `self` is a finite group or not.

EXAMPLES:

```
sage: CubicBraidGroup(6).is_finite()
False
sage: AssionGroupS(6).is_finite()
True
```

```
>>> from sage.all import *
>>> CubicBraidGroup(Integer(6)).is_finite()
False
>>> AssionGroupS(Integer(6)).is_finite()
True
```

order()

To avoid long wait-time on calculations the order will be obtained using the classical realization.

OUTPUT: cardinality of the group as integer or infinity

EXAMPLES:

```
sage: S15 = AssionGroupS(15)
sage: S15.order()
109777561863482259035023554842176139436811616256000
sage: C6 = CubicBraidGroup(6)
sage: C6.order()
+Infinity
```

```
>>> from sage.all import *
>>> S15 = AssionGroupS(Integer(15))
>>> S15.order()
109777561863482259035023554842176139436811616256000
>>> C6 = CubicBraidGroup(Integer(6))
>>> C6.order()
+Infinity
```

simple_reflections()

Return the generators of `self`.

This is only used when `self` is a finite reflection group.

EXAMPLES:

```
sage: CubicBraidGroup(3).simple_reflections()
(c0, c1)
```

```
>>> from sage.all import *
>>> CubicBraidGroup(Integer(3)).simple_reflections()
(c0, c1)
```

strands()

Return the number of strands of the braid group whose image is `self`.

OUTPUT: `Integer`

EXAMPLES:

```
sage: C4 = CubicBraidGroup(4)
sage: C4.strands()
4
```

```
>>> from sage.all import *
>>> C4 = CubicBraidGroup(Integer(4))
>>> C4.strands()
4
```

class type(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `Enum`

Enum class to select the type of the group:

- `Coxeter` – `'C'` the full cubic braid group
- `AssionS` – `'S'` finite factor group of type S considered by Assion
- `AssionU` – `'U'` finite factor group of type U considered by Assion

EXAMPLES:

```
sage: S2 = CubicBraidGroup(2, cbg_type=CubicBraidGroup.type.AssionS); S2
Assion group on 2 strands of type S
sage: U3 = CubicBraidGroup(2, cbg_type='U')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: the cbg_type must be an instance of <enum 'CubicBraidGroup.type'>
```

```
>>> from sage.all import *
>>> S2 = CubicBraidGroup(Integer(2), cbg_type=CubicBraidGroup.type.AssionS);
↪S2
Assion group on 2 strands of type S
>>> U3 = CubicBraidGroup(Integer(2), cbg_type='U')
Traceback (most recent call last):
...
TypeError: the cbg_type must be an instance of <enum 'CubicBraidGroup.type'>
```

AssionS = 'S'

AssionU = 'U'

Coxeter = 'C'

CHAPTER

FOURTEEN

INDEXED FREE GROUPS

Free groups and free abelian groups implemented using an indexed set of generators.

AUTHORS:

- Travis Scrimshaw (2013-10-16): Initial version

class sage.groups.indexed_free_group.**IndexedFreeAbelianGroup**(*indices*, *prefix*, *category=None*, ***kwds*)

Bases: *IndexedGroup*, *AbelianGroup*

An indexed free abelian group.

EXAMPLES:

```
sage: G = Groups().Commutative().free(index_set=ZZ)
sage: G
Free abelian group indexed by Integer Ring
sage: G = Groups().Commutative().free(index_set='abcde')
sage: G
Free abelian group indexed by {'a', 'b', 'c', 'd', 'e'}
```

```
>>> from sage.all import *
>>> G = Groups().Commutative().free(index_set=ZZ)
>>> G
Free abelian group indexed by Integer Ring
>>> G = Groups().Commutative().free(index_set='abcde')
>>> G
Free abelian group indexed by {'a', 'b', 'c', 'd', 'e'}
```

class Element(F, x)

Bases: IndexedFreeAbelianMonoidElement, *Element*

gen(x)

The generator indexed by x of self.

EXAMPLES:

```
sage: G = Groups().Commutative().free(index_set=ZZ)
sage: G.gen(0)
F[0]
sage: G.gen(2)
F[2]
```

```
>>> from sage.all import *
>>> G = Groups().Commutative().free(index_set=ZZ)
>>> G.gen(Integer(0))
F[0]
>>> G.gen(Integer(2))
F[2]
```

one()

Return the identity element of `self`.

EXAMPLES:

```
sage: G = Groups().Commutative().free(index_set=ZZ)
sage: G.one()
1
```

```
>>> from sage.all import *
>>> G = Groups().Commutative().free(index_set=ZZ)
>>> G.one()
1
```

class sage.groups.indexed_free_group.**IndexedFreeGroup**(*indices*, *prefix*, *category=None*, ***kwds*)

Bases: *IndexedGroup*, *Group*

An indexed free group.

EXAMPLES:

```
sage: G = Groups().free(index_set=ZZ)
sage: G
Free group indexed by Integer Ring
sage: G = Groups().free(index_set='abcde')
sage: G
Free group indexed by {'a', 'b', 'c', 'd', 'e'}
```

```
>>> from sage.all import *
>>> G = Groups().free(index_set=ZZ)
>>> G
Free group indexed by Integer Ring
>>> G = Groups().free(index_set='abcde')
>>> G
Free group indexed by {'a', 'b', 'c', 'd', 'e'}
```

class Element(F, x)

Bases: `IndexedFreeMonoidElement`

length()

Return the length of `self`.

EXAMPLES:

```
sage: G = Groups().free(index_set=ZZ)
sage: a,b,c,d,e = [G.gen(i) for i in range(5)]
sage: elt = a*c^-3*b^-2*a
```

(continues on next page)

(continued from previous page)

```
sage: elt.length()
7
sage: len(elt)
7

sage: G = Groups().free(index_set=ZZ)
sage: a,b,c,d,e = [G.gen(i) for i in range(5)]
sage: elt = a*c^-3*b^-2*a
sage: elt.length()
7
sage: len(elt)
7
```

```
>>> from sage.all import *
>>> G = Groups().free(index_set=ZZ)
>>> a,b,c,d,e = [G.gen(i) for i in range(Integer(5))]
>>> elt = a*c**-Integer(3)*b**-Integer(2)*a
>>> elt.length()
7
>>> len(elt)
7

>>> G = Groups().free(index_set=ZZ)
>>> a,b,c,d,e = [G.gen(i) for i in range(Integer(5))]
>>> elt = a*c**-Integer(3)*b**-Integer(2)*a
>>> elt.length()
7
>>> len(elt)
7
```

to_word_list()

Return `self` as a word represented as a list whose entries are the pairs `(i, s)` where `i` is the index and `s` is the sign.

EXAMPLES:

```
sage: G = Groups().free(index_set=ZZ)
sage: a,b,c,d,e = [G.gen(i) for i in range(5)]
sage: x = a*b^2*e*a^-1
sage: x.to_word_list()
[(0, 1), (1, 1), (1, 1), (4, 1), (0, -1)]
```

```
>>> from sage.all import *
>>> G = Groups().free(index_set=ZZ)
>>> a,b,c,d,e = [G.gen(i) for i in range(Integer(5))]
>>> x = a*b**Integer(2)*e*a**-Integer(1)
>>> x.to_word_list()
[(0, 1), (1, 1), (1, 1), (4, 1), (0, -1)]
```

gen(*x*)

The generator indexed by `x` of `self`.

EXAMPLES:

```
sage: G = Groups().free(index_set=ZZ)
sage: G.gen(0)
F[0]
sage: G.gen(2)
F[2]
```

```
>>> from sage.all import *
>>> G = Groups().free(index_set=ZZ)
>>> G.gen(Integer(0))
F[0]
>>> G.gen(Integer(2))
F[2]
```

one()

Return the identity element of `self`.

EXAMPLES:

```
sage: G = Groups().free(ZZ)
sage: G.one()
1
```

```
>>> from sage.all import *
>>> G = Groups().free(ZZ)
>>> G.one()
1
```

class sage.groups.indexed_free_group.**IndexedGroup**(*indices*, *prefix*, *category=None*, *names=None*, ***kwds*)

Bases: `IndexedMonoid`

Base class for free (abelian) groups whose generators are indexed by a set.

gens()

Return the group generators of `self`.

EXAMPLES:

```
sage: G = Groups.free(index_set=ZZ)
sage: G.group_generators()
Lazy family (Generator map from Integer Ring to
 Free group indexed by Integer Ring(i))_{i in Integer Ring}
sage: G = Groups().free(index_set='abcde')
sage: sorted(G.group_generators())
[F['a'], F['b'], F['c'], F['d'], F['e']]
```

```
>>> from sage.all import *
>>> G = Groups.free(index_set=ZZ)
>>> G.group_generators()
Lazy family (Generator map from Integer Ring to
 Free group indexed by Integer Ring(i))_{i in Integer Ring}
>>> G = Groups().free(index_set='abcde')
>>> sorted(G.group_generators())
[F['a'], F['b'], F['c'], F['d'], F['e']]
```

group_generators()

Return the group generators of `self`.

EXAMPLES:

```
sage: G = Groups.free(index_set=ZZ)
sage: G.group_generators()
Lazy family (Generator map from Integer Ring to
 Free group indexed by Integer Ring(i))_{i in Integer Ring}
sage: G = Groups().free(index_set='abcde')
sage: sorted(G.group_generators())
[F['a'], F['b'], F['c'], F['d'], F['e']]
```

```
>>> from sage.all import *
>>> G = Groups.free(index_set=ZZ)
>>> G.group_generators()
Lazy family (Generator map from Integer Ring to
 Free group indexed by Integer Ring(i))_{i in Integer Ring}
>>> G = Groups().free(index_set='abcde')
>>> sorted(G.group_generators())
[F['a'], F['b'], F['c'], F['d'], F['e']]
```

order()

Return the number of elements of `self`, which is ∞ unless this is the trivial group.

EXAMPLES:

```
sage: G = Groups().free(index_set=ZZ)
sage: G.order()
+Infinity
sage: G = Groups().Commutative().free(index_set='abc')
sage: G.order()
+Infinity
sage: G = Groups().Commutative().free(index_set=[])
sage: G.order()
1
```

```
>>> from sage.all import *
>>> G = Groups().free(index_set=ZZ)
>>> G.order()
+Infinity
>>> G = Groups().Commutative().free(index_set='abc')
>>> G.order()
+Infinity
>>> G = Groups().Commutative().free(index_set=[])
>>> G.order()
1
```

rank()

Return the rank of `self`.

This is the number of generators of `self`.

EXAMPLES:

```
sage: G = Groups().free(index_set=ZZ)
sage: G.rank()
+Infinity
sage: G = Groups().free(index_set='abc')
sage: G.rank()
3
sage: G = Groups().free(index_set=[])
sage: G.rank()
0
```

```
>>> from sage.all import *
>>> G = Groups().free(index_set=ZZ)
>>> G.rank()
+Infinity
>>> G = Groups().free(index_set='abc')
>>> G.rank()
3
>>> G = Groups().free(index_set=[])
>>> G.rank()
0
```

```
sage: G = Groups().Commutative().free(index_set=ZZ)
sage: G.rank()
+Infinity
sage: G = Groups().Commutative().free(index_set='abc')
sage: G.rank()
3
sage: G = Groups().Commutative().free(index_set=[])
sage: G.rank()
0
```

```
>>> from sage.all import *
>>> G = Groups().Commutative().free(index_set=ZZ)
>>> G.rank()
+Infinity
>>> G = Groups().Commutative().free(index_set='abc')
>>> G.rank()
3
>>> G = Groups().Commutative().free(index_set=[])
>>> G.rank()
0
```

CHAPTER

FIFTEEN

ARTIN GROUPS

Artin groups are implemented as a particular case of finitely presented groups. For finite-type Artin groups, there is a specific left normal form using the Garside structure associated to the lift the long element of the corresponding Coxeter group.

AUTHORS:

- Travis Scrimshaw (2018-02-05): Initial version
- Travis Scrimshaw (2025-01-30): Allowed general construction; implemented general Burau representation

Todo

Implement affine type Artin groups with their well-known embeddings into the classical braid group.

class sage.groups.artin.**ArtinGroup**(*coxeter_matrix*, *names*)

Bases: `UniqueRepresentation`, `FinitelyPresentedGroup`

An Artin group.

Fix an index set I. Let $M = (m_{ij})_{i,j \in I}$ be a `Coxeter matrix`. An *Artin group* is a group A_M that has a presentation given by generators $\{s_i \mid i \in I\}$ and relations

$$\underbrace{s_i s_j s_i \cdots}_{m_{ij}} = \underbrace{s_j s_i s_j \cdots}_{m_{ji} \text{ factors}}$$

for all $i, j \in I$ with the usual convention that $m_{ij} = \infty$ implies no relation between s_i and s_j. There is a natural corresponding Coxeter group W_M by imposing the additional relations $s_i^2 = 1$ for all $i \in I$. Furthermore, there is a natural section of W_M by sending a reduced word $s_{i_1} \cdots s_{i_\ell} \mapsto s_{i_1} \cdots s_{i_\ell}$.

Artin groups A_M are classified based on the Coxeter data:

- A_M is of *finite type* or *spherical* if W_M is finite;
- A_M is of *affine type* if W_M is of affine type;
- A_M is of *large type* if $m_{ij} \geq 4$ for all $i, j \in I$;
- A_M is of *extra-large type* if $m_{ij} \geq 5$ for all $i, j \in I$;
- A_M is *right-angled* if $m_{ij} \in \{2, \infty\}$ for all $i, j \in I$.

Artin groups are conjectured to have many nice properties:

- Artin groups are torsion free.
- Finite type Artin groups have $Z(A_M) = \mathbf{Z}$ and infinite type Artin groups have trivial center.
- Artin groups have solvable word problems.

- H_{W_M}/W_M is a $K(A_M, 1)$-space, where H_W is the hyperplane complement of the Coxeter group W acting on $\mathbf{C}^n$.

These conjectures are known when the Artin group is finite type and a number of other cases. See, e.g., [GP2012] and references therein.

INPUT:

- `coxeter_data` – data defining a Coxeter matrix
- `names` – string or list/tuple/iterable of strings (default: `'s'`); the generator names or name prefix

EXAMPLES:

```
sage: A.<a,b,c> = ArtinGroup(['B',3]);  A                                      #␣
↪needs sage.rings.number_field
Artin group of type ['B', 3]
sage: ArtinGroup(['B',3])                                                      #␣
↪needs sage.rings.number_field
Artin group of type ['B', 3]
```

```
>>> from sage.all import *
>>> A = ArtinGroup(['B',Integer(3)], names=('a', 'b', 'c',)); (a, b, c,) = A._
↪first_ngens(3);  A                                       # needs sage.rings.
↪number_field
Artin group of type ['B', 3]
>>> ArtinGroup(['B',Integer(3)])                                               ␣
↪     # needs sage.rings.number_field
Artin group of type ['B', 3]
```

The input must always include the Coxeter data, but the `names` can either be a string representing the prefix of the names or the explicit names of the generators. Otherwise the default prefix of `'s'` is used:

```
sage: ArtinGroup(['B',2]).generators()                                         #␣
↪needs sage.rings.number_field
(s1, s2)
sage: ArtinGroup(['B',2], 'g').generators()                                    #␣
↪needs sage.rings.number_field
(g1, g2)
sage: ArtinGroup(['B',2], 'x,y').generators()                                  #␣
↪needs sage.rings.number_field
(x, y)
```

```
>>> from sage.all import *
>>> ArtinGroup(['B',Integer(2)]).generators()                                  ␣
↪     # needs sage.rings.number_field
(s1, s2)
>>> ArtinGroup(['B',Integer(2)], 'g').generators()                             ␣
↪     # needs sage.rings.number_field
(g1, g2)
>>> ArtinGroup(['B',Integer(2)], 'x,y').generators()                           ␣
↪     # needs sage.rings.number_field
(x, y)
```

REFERENCES:

- Wikipedia article Artin_group

See also

RightAngledArtinGroup

Element
alias of *ArtinGroupElement*

an_element()
Return an element of `self`.

EXAMPLES:

```
sage: A = ArtinGroup(['B',2])                                               #␣
↪needs sage.rings.number_field
sage: A.an_element()                                                        #␣
↪needs sage.rings.number_field
s1
```

```
>>> from sage.all import *
>>> A = ArtinGroup(['B',Integer(2)])                                         ␣
↪     # needs sage.rings.number_field
>>> A.an_element()                                                        #␣
↪needs sage.rings.number_field
s1
```

as_permutation_group()
Return an isomorphic permutation group.

This raises a `ValueError` error since Artin groups are infinite and have no corresponding permutation group.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.as_permutation_group()
Traceback (most recent call last):
...
ValueError: the group is infinite

sage: A = ArtinGroup(['D',4], 'g')                                          #␣
↪needs sage.rings.number_field
sage: A.as_permutation_group()                                              #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
ValueError: the group is infinite
```

```
>>> from sage.all import *
>>> Gamma = graphs.CycleGraph(Integer(5))
>>> G = RightAngledArtinGroup(Gamma)
>>> G.as_permutation_group()
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: the group is infinite

>>> A = ArtinGroup(['D',Integer(4)], 'g')                                    ␣
↪     # needs sage.rings.number_field
>>> A.as_permutation_group()                                               #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
ValueError: the group is infinite
```

cardinality()

Return the number of elements of `self`.

OUTPUT: infinity

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.cardinality()
+Infinity

sage: A = ArtinGroup(['A',1])                                               #␣
↪needs sage.rings.number_field
sage: A.cardinality()                                                       #␣
↪needs sage.rings.number_field
+Infinity
```

```
>>> from sage.all import *
>>> Gamma = graphs.CycleGraph(Integer(5))
>>> G = RightAngledArtinGroup(Gamma)
>>> G.cardinality()
+Infinity

>>> A = ArtinGroup(['A',Integer(1)])                                         ␣
↪     # needs sage.rings.number_field
>>> A.cardinality()                                                        #␣
↪needs sage.rings.number_field
+Infinity
```

coxeter_group()

Return the Coxeter group of `self`.

EXAMPLES:

```
sage: A = ArtinGroup(['D',4])                                               #␣
↪needs sage.rings.number_field
sage: A.coxeter_group()                                                     #␣
↪needs sage.rings.number_field
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3 2 2]
[3 1 3 3]
[2 3 1 2]
[2 3 2 1]
```

```
>>> from sage.all import *
>>> A = ArtinGroup(['D',Integer(4)])                                    ␣
↪     # needs sage.rings.number_field
>>> A.coxeter_group()                                                 #␣
↪needs sage.rings.number_field
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3 2 2]
[3 1 3 3]
[2 3 1 2]
[2 3 2 1]
```

coxeter_matrix()

Return the Coxeter matrix of `self`.

EXAMPLES:

```
sage: A = ArtinGroup(['B',3])                                          #␣
↪needs sage.rings.number_field
sage: A.coxeter_matrix()                                               #␣
↪needs sage.rings.number_field
[1 3 2]
[3 1 4]
[2 4 1]
```

```
>>> from sage.all import *
>>> A = ArtinGroup(['B',Integer(3)])                                    ␣
↪     # needs sage.rings.number_field
>>> A.coxeter_matrix()                                                #␣
↪needs sage.rings.number_field
[1 3 2]
[3 1 4]
[2 4 1]
```

coxeter_type()

Return the Coxeter type of `self`.

EXAMPLES:

```
sage: A = ArtinGroup(['D',4])                                          #␣
↪needs sage.rings.number_field
sage: A.coxeter_type()                                                 #␣
↪needs sage.rings.number_field
Coxeter type of ['D', 4]
```

```
>>> from sage.all import *
>>> A = ArtinGroup(['D',Integer(4)])                                    ␣
↪     # needs sage.rings.number_field
>>> A.coxeter_type()                                                  #␣
↪needs sage.rings.number_field
Coxeter type of ['D', 4]
```

index_set()

Return the index set of `self`.

OUTPUT: tuple

EXAMPLES:

```
sage: A = ArtinGroup(['E',7])                                               #␣
↪needs sage.rings.number_field
sage: A.index_set()                                                         #␣
↪needs sage.rings.number_field
(1, 2, 3, 4, 5, 6, 7)
```

```
>>> from sage.all import *
>>> A = ArtinGroup(['E',Integer(7)])                                         ␣
↪      # needs sage.rings.number_field
>>> A.index_set()                                                         #␣
↪needs sage.rings.number_field
(1, 2, 3, 4, 5, 6, 7)
```

order()

Return the number of elements of `self`.

OUTPUT: infinity

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.cardinality()
+Infinity

sage: A = ArtinGroup(['A',1])                                               #␣
↪needs sage.rings.number_field
sage: A.cardinality()                                                       #␣
↪needs sage.rings.number_field
+Infinity
```

```
>>> from sage.all import *
>>> Gamma = graphs.CycleGraph(Integer(5))
>>> G = RightAngledArtinGroup(Gamma)
>>> G.cardinality()
+Infinity

>>> A = ArtinGroup(['A',Integer(1)])                                         ␣
↪      # needs sage.rings.number_field
>>> A.cardinality()                                                       #␣
↪needs sage.rings.number_field
+Infinity
```

some_elements()

Return a list of some elements of `self`.

EXAMPLES:

```
sage: A = ArtinGroup(['B',3])                                               #␣
↪needs sage.rings.number_field
sage: A.some_elements()                                                     #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.number_field
[s1, s1*s2*s3, (s1*s2*s3)^3]
```

```
>>> from sage.all import *
>>> A = ArtinGroup(['B',Integer(3)])                                          ␣
↪      # needs sage.rings.number_field
>>> A.some_elements()                                                        #␣
↪needs sage.rings.number_field
[s1, s1*s2*s3, (s1*s2*s3)^3]
```

class sage.groups.artin.**ArtinGroupElement**(*parent*, *x*, *check=True*)

Bases: *FinitelyPresentedGroupElement*

An element of an Artin group.

It is a particular case of element of a finitely presented group.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: A.<s1,s2,s3> = ArtinGroup(['B',3])
sage: A
Artin group of type ['B', 3]
sage: s1 * s2 / s3 / s2
s1*s2*s3^-1*s2^-1
sage: A((1, 2, -3, -2))
s1*s2*s3^-1*s2^-1
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> A = ArtinGroup(['B',Integer(3)], names=('s1', 's2', 's3',)); (s1, s2, s3,) =␣
↪A._first_ngens(3)
>>> A
Artin group of type ['B', 3]
>>> s1 * s2 / s3 / s2
s1*s2*s3^-1*s2^-1
>>> A((Integer(1), Integer(2), -Integer(3), -Integer(2)))
s1*s2*s3^-1*s2^-1
```

burau_matrix(*var='t'*)

Return the Burau matrix of the Artin group element.

Following [BQ2024], the (generalized) Burau representation of an Artin group is defined by deforming the reflection representation of the corresponding Coxeter group. However, we substitute $q \mapsto -t$ from [BQ2024] to match one of the unitary (reduced) Burau representations of the braid group (see *sage.groups.braid.Braid.burau_matrix()* for details.)

More precisely, let $(m_{ij})_{i,j \in I}$ be the `Coxeter matrix`. Then the action is given on the basis $(\alpha_1, \ldots \alpha_n)$ (corresponding to the reflection representation of the corresponding `Coxeter group`) by

$$\sigma_i(\alpha_j) = \alpha_j - \langle \alpha_i, \alpha_j \rangle_q \alpha_i, \qquad \text{where } \langle \alpha_i, \alpha_j \rangle_q := \begin{cases} 1 + t^2 & \text{if } i = j, \\ -2t\cos(\pi/m_{ij}) & \text{if } i \neq j. \end{cases}.$$

By convention $\cos(\pi/\infty) = 1$. Note that the inverse of the generators act by $\sigma_i^{-1}(\alpha_j) = \alpha_j - q^{-2}\langle \alpha_j, \alpha_i \rangle_q \alpha_i$.

INPUT:

- `var` – string (default: `'t'`); the name of the variable in the entries of the matrix

OUTPUT:

The Burau matrix of the Artin group element over the Laurent polynomial ring in the variable `var`.

EXAMPLES:

```
sage: A.<s1,s2,s3> = ArtinGroup(['B',3])
sage: B1 = s1.burau_matrix()
sage: B2 = s2.burau_matrix()
sage: B3 = s3.burau_matrix()
sage: [B1, B2, B3]
[
[-t^2    t    0]  [   1    0    0]  [   1    0    0]
[   0    1    0]  [   t -t^2  a*t]  [   0    1    0]
[   0    0    1], [   0    0    1], [   0  a*t -t^2]
]
sage: B1 * B2 * B1 == B2 * B1 * B2
True
sage: B2 * B3 * B2 * B3 == B3 * B2 * B3 * B2
True
sage: B1 * B3 == B3 * B1
True
sage: (~s1).burau_matrix() * B1 == 1
True
```

```
>>> from sage.all import *
>>> A = ArtinGroup(['B',Integer(3)], names=('s1', 's2', 's3',)); (s1, s2, s3,
↪) = A._first_ngens(3)
>>> B1 = s1.burau_matrix()
>>> B2 = s2.burau_matrix()
>>> B3 = s3.burau_matrix()
>>> [B1, B2, B3]
[
[-t^2    t    0]  [   1    0    0]  [   1    0    0]
[   0    1    0]  [   t -t^2  a*t]  [   0    1    0]
[   0    0    1], [   0    0    1], [   0  a*t -t^2]
]
>>> B1 * B2 * B1 == B2 * B1 * B2
True
>>> B2 * B3 * B2 * B3 == B3 * B2 * B3 * B2
True
>>> B1 * B3 == B3 * B1
True
>>> (~s1).burau_matrix() * B1 == Integer(1)
True
```

We verify the example in Theorem 4.1 of [BQ2024]:

```
sage: A.<s1,s2,s3,s4> = ArtinGroup(['A', 3, 1])
sage: [g.burau_matrix() for g in A.gens()]
[
```

(continues on next page)

(continued from previous page)

```
[-t^2    t    0    t]  [   1    0    0    0]  [   1    0    0    0]
[   0    1    0    0]  [   t -t^2    t    0]  [   0    1    0    0]
[   0    0    1    0]  [   0    0    1    0]  [   0    t -t^2    t]
[   0    0    0    1], [   0    0    0    1], [   0    0    0    1],

[   1    0    0    0]
[   0    1    0    0]
[   0    0    1    0]
[   t    0    t -t^2]
]
sage: a = s3^2 * s4 * s3 * s2 *s1 * ~s3 * s4 * s3 * s2 * s1^-2 * s4
sage: b = s1^2 * ~s2 * s4 * s1 * ~s3 * s2 * ~s4 * s3 * s1 * s4 * s1 * ~s2 *␣
↪s4^-2 * s3
sage: alpha = a * s3 * ~a
sage: beta = b * s2 * ~b
sage: elm = alpha * beta * ~alpha * ~beta
sage: print(elm.Tietze())
(3, 3, 4, 3, 2, 1, -3, 4, 3, 2, -1, -1, 4, 3, -4, 1, 1, -2, -3,
 -4, 3, -1, -2, -3, -4, -3, -3, 1, 1, -2, 4, 1, -3, 2, -4, 3, 1,
 4, 1, -2, -4, -4, 3, 2, -3, 4, 4, 2, -1, -4, -1, -3, 4, -2, 3,
 -1, -4, 2, -1, -1, 3, 3, 4, 3, 2, 1, -3, 4, 3, 2, -1, -1, 4,
 -3, -4, 1, 1, -2, -3, -4, 3, -1, -2, -3, -4, -3, -3, 1, 1, -2,
 4, 1, -3, 2, -4, 3, 1, 4, 1, -2, -4, -4, 3, -2, -3, 4, 4, 2,
 -1, -4, -1, -3, 4, -2, 3, -1, -4, 2, -1, -1)
sage: elm.burau_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> A = ArtinGroup(['A', Integer(3), Integer(1)], names=('s1', 's2', 's3', 's4
↪',)); (s1, s2, s3, s4,) = A._first_ngens(4)
>>> [g.burau_matrix() for g in A.gens()]
[
[-t^2    t    0    t]  [   1    0    0    0]  [   1    0    0    0]
[   0    1    0    0]  [   t -t^2    t    0]  [   0    1    0    0]
[   0    0    1    0]  [   0    0    1    0]  [   0    t -t^2    t]
[   0    0    0    1], [   0    0    0    1], [   0    0    0    1],
<BLANKLINE>
[   1    0    0    0]
[   0    1    0    0]
[   0    0    1    0]
[   t    0    t -t^2]
]
>>> a = s3**Integer(2) * s4 * s3 * s2 *s1 * ~s3 * s4 * s3 * s2 * s1**-
↪Integer(2) * s4
>>> b = s1**Integer(2) * ~s2 * s4 * s1 * ~s3 * s2 * ~s4 * s3 * s1 * s4 * s1 *␣
↪~s2 * s4**-Integer(2) * s3
>>> alpha = a * s3 * ~a
>>> beta = b * s2 * ~b
>>> elm = alpha * beta * ~alpha * ~beta
```

(continues on next page)

(continued from previous page)

```
>>> print(elm.Tietze())
(3, 3, 4, 3, 2, 1, -3, 4, 3, 2, -1, -1, 4, 3, -4, 1, 1, -2, -3,
 -4, 3, -1, -2, -3, -4, -3, -3, 1, 1, -2, 4, 1, -3, 2, -4, 3, 1,
 4, 1, -2, -4, -4, 3, 2, -3, 4, 4, 2, -1, -4, -1, -3, 4, -2, 3,
 -1, -4, 2, -1, -1, 3, 3, 4, 3, 2, 1, -3, 4, 3, 2, -1, -1, 4,
 -3, -4, 1, 1, -2, -3, -4, 3, -1, -2, -3, -4, -3, -3, 1, 1, -2,
 4, 1, -3, 2, -4, 3, 1, 4, 1, -2, -4, -4, 3, -2, -3, 4, 4, 2,
 -1, -4, -1, -3, 4, -2, 3, -1, -4, 2, -1, -1)
>>> elm.burau_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

Next, we show `elm` is not the identity by using the embedding of the affine braid group $\widetilde{B}_n \to B_{n+1}$:

```
sage: B.<t1,t2,t3,t4> = BraidGroup(5)
sage: D = t1 * t2 * t3 * t4^2
sage: t0 = D * t3 * ~D
sage: t0*t1*t0 == t1*t0*t1
True
sage: t0*t2 == t2*t0
True
sage: t0*t3*t0 == t3*t0*t3
True
sage: T = [t0, t1, t2, t3]
sage: emb = B.prod(T[i-1] if i > 0 else ~T[-i-1] for i in elm.Tietze())
sage: emb.is_one()
False
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(5), names=('t1', 't2', 't3', 't4',)); (t1, t2, t3,␣
↪t4,) = B._first_ngens(4)
>>> D = t1 * t2 * t3 * t4**Integer(2)
>>> t0 = D * t3 * ~D
>>> t0*t1*t0 == t1*t0*t1
True
>>> t0*t2 == t2*t0
True
>>> t0*t3*t0 == t3*t0*t3
True
>>> T = [t0, t1, t2, t3]
>>> emb = B.prod(T[i-Integer(1)] if i > Integer(0) else ~T[-i-Integer(1)] for␣
↪i in elm.Tietze())
>>> emb.is_one()
False
```

Since the Burau representation does not respect the group embedding, the corresponding B_5 element's Burau matrix is not the identity (Bigelow gave an example of the representation not being faithful for B_5, but it is still open for B_4):

```
sage: emb.burau_matrix() != 1
```

(continues on next page)

(continued from previous page)

```
True
```

```
>>> from sage.all import *
>>> emb.burau_matrix() != Integer(1)
True
```

We also verify the result using the elements in [BQ2024] Remark 4.2:

```
sage: ap = s3 * s1 * s2 * s1 * ~s3 * s4 * s2 * s3 * s2 * ~s3 * s1^-2 * s4
sage: bp = s1 * ~s4 * s1^2 * s3^-2 * ~s2 * s4 * s1 * ~s3 * s2 * ~s4 * s3 * s1␣
↪* s4 * s1 * ~s2 * s4^-2 * s3
sage: alpha = ap * s3 * ~ap
sage: beta = bp * s2 * ~bp
sage: elm = alpha * beta * ~alpha * ~beta
sage: elm.burau_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> ap = s3 * s1 * s2 * s1 * ~s3 * s4 * s2 * s3 * s2 * ~s3 * s1**-Integer(2)␣
↪* s4
>>> bp = s1 * ~s4 * s1**Integer(2) * s3**-Integer(2) * ~s2 * s4 * s1 * ~s3 *␣
↪s2 * ~s4 * s3 * s1 * s4 * s1 * ~s2 * s4**-Integer(2) * s3
>>> alpha = ap * s3 * ~ap
>>> beta = bp * s2 * ~bp
>>> elm = alpha * beta * ~alpha * ~beta
>>> elm.burau_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

REFERENCES:

- [BQ2024]

coxeter_group_element(*W=None*)

Return the corresponding Coxeter group element under the natural projection.

INPUT:

- `W` – (default: `self.parent().coxeter_group()`) the image Coxeter group

OUTPUT: an element of the Coxeter group `W`

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: B.<s1,s2,s3> = ArtinGroup(['B',3])
sage: b = s1 * s2 / s3 / s2
sage: b1 = b.coxeter_group_element(); b1
[ 1 -1  0]
[ 2 -1  0]
```

(continues on next page)

(continued from previous page)

```
[ a -a  1]
sage: b.coxeter_group_element().reduced_word()
[1, 2, 3, 2]
sage: A.<s1,s2,s3> = ArtinGroup(['A',3])
sage: c = s1 * s2 *s3
sage: c1 = c.coxeter_group_element(); c1
[4, 1, 2, 3]
sage: c1.reduced_word()
[3, 2, 1]
sage: c.coxeter_group_element(W=SymmetricGroup(4))
(1,4,3,2)
sage: A.<s1,s2,s3> = BraidGroup(4)
sage: c = s1 * s2 * s3^-1
sage: c0 = c.coxeter_group_element(); c0
[4, 1, 2, 3]
sage: c1 = c.coxeter_group_element(W=SymmetricGroup(4)); c1
(1,4,3,2)
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> B = ArtinGroup(['B',Integer(3)], names=('s1', 's2', 's3',)); (s1, s2, s3,
↪) = B._first_ngens(3)
>>> b = s1 * s2 / s3 / s2
>>> b1 = b.coxeter_group_element(); b1
[ 1 -1  0]
[ 2 -1  0]
[ a -a  1]
>>> b.coxeter_group_element().reduced_word()
[1, 2, 3, 2]
>>> A = ArtinGroup(['A',Integer(3)], names=('s1', 's2', 's3',)); (s1, s2, s3,
↪) = A._first_ngens(3)
>>> c = s1 * s2 *s3
>>> c1 = c.coxeter_group_element(); c1
[4, 1, 2, 3]
>>> c1.reduced_word()
[3, 2, 1]
>>> c.coxeter_group_element(W=SymmetricGroup(Integer(4)))
(1,4,3,2)
>>> A = BraidGroup(Integer(4), names=('s1', 's2', 's3',)); (s1, s2, s3,) = A._
↪first_ngens(3)
>>> c = s1 * s2 * s3**-Integer(1)
>>> c0 = c.coxeter_group_element(); c0
[4, 1, 2, 3]
>>> c1 = c.coxeter_group_element(W=SymmetricGroup(Integer(4))); c1
(1,4,3,2)
```

From an element of the Coxeter group it is possible to recover the image by the standard section to the Artin group:

```
sage: # needs sage.rings.number_field
sage: B(b1)
s1*s2*s3*s2
```

(continues on next page)

(continued from previous page)

```
sage: A(c0)
s1*s2*s3
sage: A(c0) == A(c1)
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> B(b1)
s1*s2*s3*s2
>>> A(c0)
s1*s2*s3
>>> A(c0) == A(c1)
True
```

exponent_sum()

Return the exponent sum of `self`.

OUTPUT: integer

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: A = ArtinGroup(['E',6])
sage: b = A([1, 4, -3, 2])
sage: b.exponent_sum()
2
sage: b = A([])
sage: b.exponent_sum()
0

sage: B = BraidGroup(5)
sage: b = B([1, 4, -3, 2])
sage: b.exponent_sum()
2
sage: b = B([])
sage: b.exponent_sum()
0
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> A = ArtinGroup(['E',Integer(6)])
>>> b = A([Integer(1), Integer(4), -Integer(3), Integer(2)])
>>> b.exponent_sum()
2
>>> b = A([])
>>> b.exponent_sum()
0

>>> B = BraidGroup(Integer(5))
>>> b = B([Integer(1), Integer(4), -Integer(3), Integer(2)])
>>> b.exponent_sum()
2
```

(continues on next page)

(continued from previous page)

```
>>> b = B([])
>>> b.exponent_sum()
0
```

class sage.groups.artin.**FiniteTypeArtinGroup**(*coxeter_matrix*, *names*)

Bases: *ArtinGroup*

A finite-type Artin group.

An Artin group is *finite-type* or *spherical* if the corresponding Coxeter group is finite. Finite type Artin groups are known to be torsion free, have a Garside structure given by Δ (see *delta()*) and have a center generated by Δ.

See also

ArtinGroup

EXAMPLES:

```
sage: ArtinGroup(['E',7])                                               #␣
↪needs sage.rings.number_field
Artin group of type ['E', 7]
```

```
>>> from sage.all import *
>>> ArtinGroup(['E',Integer(7)])                                        ␣
↪       # needs sage.rings.number_field
Artin group of type ['E', 7]
```

Since the word problem for finite-type Artin groups is solvable, their Cayley graph can be locally obtained as follows (see Issue #16059):

```
sage: def ball(group, radius):
....:     ret = set()
....:     ret.add(group.one())
....:     for length in range(1, radius):
....:         for w in Words(alphabet=group.gens(), length=length):
....:              ret.add(prod(w))
....:     return ret
sage: A = ArtinGroup(['B',3])                                           #␣
↪needs sage.rings.number_field
sage: GA = A.cayley_graph(elements=ball(A, 4), generators=A.gens()); GA     #␣
↪needs sage.rings.number_field
Digraph on 32 vertices
```

```
>>> from sage.all import *
>>> def ball(group, radius):
...     ret = set()
...     ret.add(group.one())
...     for length in range(Integer(1), radius):
...         for w in Words(alphabet=group.gens(), length=length):
...              ret.add(prod(w))
...     return ret
>>> A = ArtinGroup(['B',Integer(3)])                                    ␣
```

(continues on next page)

(continued from previous page)

```
↪     # needs sage.rings.number_field
>>> GA = A.cayley_graph(elements=ball(A, Integer(4)), generators=A.gens()); GA   ␣
↪     # needs sage.rings.number_field
Digraph on 32 vertices
```

Since the Artin group has nontrivial relations, this graph contains less vertices than the one associated to the free group (which is a tree):

```
sage: F = FreeGroup(3)
sage: GF = F.cayley_graph(elements=ball(F, 4), generators=F.gens()); GF         #␣
↪needs sage.combinat
Digraph on 40 vertices
```

```
>>> from sage.all import *
>>> F = FreeGroup(Integer(3))
>>> GF = F.cayley_graph(elements=ball(F, Integer(4)), generators=F.gens()); GF   ␣
↪     # needs sage.combinat
Digraph on 40 vertices
```

Element

alias of *FiniteTypeArtinGroupElement*

delta()

Return the Δ element of `self`.

EXAMPLES:

```
sage: A = ArtinGroup(['B',3])                                                   #␣
↪needs sage.rings.number_field
sage: A.delta()                                                                 #␣
↪needs sage.rings.number_field
s3*(s2*s3*s1)^2*s2*s1

sage: A = ArtinGroup(['G',2])                                                   #␣
↪needs sage.rings.number_field
sage: A.delta()                                                                 #␣
↪needs sage.rings.number_field
(s2*s1)^3

sage: B = BraidGroup(5)
sage: B.delta()
s0*s1*s2*s3*s0*s1*s2*s0*s1*s0
```

```
>>> from sage.all import *
>>> A = ArtinGroup(['B',Integer(3)])                                            ␣
↪     # needs sage.rings.number_field
>>> A.delta()                                                                 #␣
↪needs sage.rings.number_field
s3*(s2*s3*s1)^2*s2*s1

>>> A = ArtinGroup(['G',Integer(2)])                                            ␣
↪     # needs sage.rings.number_field
```

(continues on next page)

(continued from previous page)

```
>>> A.delta()                                                              #␣
↪needs sage.rings.number_field
(s2*s1)^3

>>> B = BraidGroup(Integer(5))
>>> B.delta()
s0*s1*s2*s3*s0*s1*s2*s0*s1*s0
```

class sage.groups.artin.**FiniteTypeArtinGroupElement**(*parent*, *x*, *check=True*)

Bases: `ArtinGroupElement`

An element of a finite-type Artin group.

left_normal_form()

Return the left normal form of `self`.

OUTPUT:

A tuple of simple generators in the left normal form. The first element is a power of Δ, and the rest are elements of the natural section lift from the corresponding Coxeter group.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: A = ArtinGroup(['B',3])
sage: A([1]).left_normal_form()
(1, s1)
sage: A([-1]).left_normal_form()
(s1^-1*(s2^-1*s1^-1*s3^-1)^2*s2^-1*s3^-1, s3*(s2*s3*s1)^2*s2)
sage: A([1, 2, 2, 1, 2]).left_normal_form()
(1, s1*s2*s1, s2*s1)
sage: A([3, 3, -2]).left_normal_form()
(s1^-1*(s2^-1*s1^-1*s3^-1)^2*s2^-1*s3^-1,
 s3*s1*s2*s3*s2*s1, s3, s3*s2*s3)
sage: A([1, 2, 3, -1, 2, -3]).left_normal_form()
(s1^-1*(s2^-1*s1^-1*s3^-1)^2*s2^-1*s3^-1,
 (s3*s1*s2)^2*s1, s1*s2*s3*s2)
sage: A([1,2,1,3,2,1,3,2,3,3,2,3,1,2,3,1,2,3,1,2]).left_normal_form()
((s3*(s2*s3*s1)^2*s2*s1)^2, s3*s2)

sage: B = BraidGroup(4)
sage: b = B([1, 2, 3, -1, 2, -3])
sage: b.left_normal_form()
(s0^-1*s1^-1*s0^-1*s2^-1*s1^-1*s0^-1, s0*s1*s2*s1*s0, s0*s2*s1)
sage: c = B([1])
sage: c.left_normal_form()
(1, s0)
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> A = ArtinGroup(['B',Integer(3)])
>>> A([Integer(1)]).left_normal_form()
(1, s1)
>>> A([-Integer(1)]).left_normal_form()
```

(continues on next page)

(continued from previous page)

```
(s1^-1*(s2^-1*s1^-1*s3^-1)^2*s2^-1*s3^-1, s3*(s2*s3*s1)^2*s2)
>>> A([Integer(1), Integer(2), Integer(2), Integer(1), Integer(2)]).left_
↪normal_form()
(1, s1*s2*s1, s2*s1)
>>> A([Integer(3), Integer(3), -Integer(2)]).left_normal_form()
(s1^-1*(s2^-1*s1^-1*s3^-1)^2*s2^-1*s3^-1,
 s3*s1*s2*s3*s2*s1, s3, s3*s2*s3)
>>> A([Integer(1), Integer(2), Integer(3), -Integer(1), Integer(2), -
↪Integer(3)]).left_normal_form()
(s1^-1*(s2^-1*s1^-1*s3^-1)^2*s2^-1*s3^-1,
 (s3*s1*s2)^2*s1, s1*s2*s3*s2)
>>> A([Integer(1),Integer(2),Integer(1),Integer(3),Integer(2),Integer(1),
↪Integer(3),Integer(2),Integer(3),Integer(3),Integer(2),Integer(3),
↪Integer(1),Integer(2),Integer(3),Integer(1),Integer(2),Integer(3),
↪Integer(1),Integer(2)]).left_normal_form()
((s3*(s2*s3*s1)^2*s2*s1)^2, s3*s2)

>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1), Integer(2), Integer(3), -Integer(1), Integer(2), -
↪Integer(3)])
>>> b.left_normal_form()
(s0^-1*s1^-1*s0^-1*s2^-1*s1^-1*s0^-1, s0*s1*s2*s1*s0, s0*s2*s1)
>>> c = B([Integer(1)])
>>> c.left_normal_form()
(1, s0)
```

CHAPTER SIXTEEN

RIGHT-ANGLED ARTIN GROUPS

A *right-angled Artin group* (often abbreviated as RAAG) is a group which has a presentation whose only relations are commutators between generators. These are also known as graph groups, since they are (uniquely) encoded by (simple) graphs, or partially commutative groups.

AUTHORS:

- Travis Scrimshaw (2013-09-01): Initial version
- Travis Scrimshaw (2018-02-05): Made compatible with `ArtinGroup`

class sage.groups.raag.**CohomologyRAAG**(R, A)

Bases: `CombinatorialFreeModule`

The cohomology ring of a right-angled Artin group.

The cohomology ring of a right-angled Artin group A, defined by the graph G, with coefficients in a field F is isomorphic to the exterior algebra of F^N, where N is the number of vertices in G, modulo the quadratic relations $e_i \wedge e_j = 0$ if and only if (i, j) is an edge in G. This algebra is sometimes also known as the Cartier-Foata algebra.

REFERENCES:

- [CQ2019]

class Element

Bases: `CohomologyRAAGElement`

An element in the cohomology ring of a right-angled Artin group.

algebra_generators()

Return the algebra generators of `self`.

EXAMPLES:

```
sage: C4 = graphs.CycleGraph(4)
sage: A = groups.misc.RightAngledArtin(C4)
sage: H = A.cohomology()
sage: H.algebra_generators()
Finite family {0: e0, 1: e1, 2: e2, 3: e3}
```

```
>>> from sage.all import *
>>> C4 = graphs.CycleGraph(Integer(4))
>>> A = groups.misc.RightAngledArtin(C4)
>>> H = A.cohomology()
>>> H.algebra_generators()
Finite family {0: e0, 1: e1, 2: e2, 3: e3}
```

degree_on_basis(*I*)

Return the degree on the basis element `clique`.

EXAMPLES:

```
sage: C4 = graphs.CycleGraph(4)
sage: A = groups.misc.RightAngledArtin(C4)
sage: H = A.cohomology()
sage: sorted([H.degree_on_basis(I) for I in H.basis().keys()])
[0, 1, 1, 1, 1, 2, 2]
```

```
>>> from sage.all import *
>>> C4 = graphs.CycleGraph(Integer(4))
>>> A = groups.misc.RightAngledArtin(C4)
>>> H = A.cohomology()
>>> sorted([H.degree_on_basis(I) for I in H.basis().keys()])
[0, 1, 1, 1, 1, 2, 2]
```

gen(*i*)

Return the `i`-th standard generator of the algebra `self`.

This corresponds to the `i`-th vertex in the graph (under a fixed ordering of the vertices).

EXAMPLES:

```
sage: C4 = graphs.CycleGraph(4)
sage: A = groups.misc.RightAngledArtin(C4)
sage: H = A.cohomology()
sage: H.gen(0)
e0
sage: H.gen(1)
e1
```

```
>>> from sage.all import *
>>> C4 = graphs.CycleGraph(Integer(4))
>>> A = groups.misc.RightAngledArtin(C4)
>>> H = A.cohomology()
>>> H.gen(Integer(0))
e0
>>> H.gen(Integer(1))
e1
```

gens()

Return the generators of `self` (as an algebra).

EXAMPLES:

```
sage: C4 = graphs.CycleGraph(4)
sage: A = groups.misc.RightAngledArtin(C4)
sage: H = A.cohomology()
sage: H.gens()
(e0, e1, e2, e3)
```

```
>>> from sage.all import *
>>> C4 = graphs.CycleGraph(Integer(4))
>>> A = groups.misc.RightAngledArtin(C4)
>>> H = A.cohomology()
>>> H.gens()
(e0, e1, e2, e3)
```

ngens()

Return the number of algebra generators of `self`.

EXAMPLES:

```
sage: C4 = graphs.CycleGraph(4)
sage: A = groups.misc.RightAngledArtin(C4)
sage: H = A.cohomology()
sage: H.ngens()
4
```

```
>>> from sage.all import *
>>> C4 = graphs.CycleGraph(Integer(4))
>>> A = groups.misc.RightAngledArtin(C4)
>>> H = A.cohomology()
>>> H.ngens()
4
```

one_basis()

Return the basis element indexing 1 of `self`.

EXAMPLES:

```
sage: C4 = graphs.CycleGraph(4)
sage: A = groups.misc.RightAngledArtin(C4)
sage: H = A.cohomology()
sage: H.one_basis()
()
```

```
>>> from sage.all import *
>>> C4 = graphs.CycleGraph(Integer(4))
>>> A = groups.misc.RightAngledArtin(C4)
>>> H = A.cohomology()
>>> H.one_basis()
()
```

class sage.groups.raag.**RightAngledArtinGroup**(G, *names*)

Bases: *ArtinGroup*

The right-angled Artin group defined by a graph G.

Let $\Gamma = \{V(\Gamma), E(\Gamma)\}$ be a simple graph. A *right-angled Artin group* (commonly abbreviated as RAAG) is the group

$$A_\Gamma = \langle g_v : v \in V(\Gamma) \mid [g_u, g_v] \text{ if } \{u, v\} \notin E(\Gamma) \rangle.$$

These are sometimes known as graph groups or partially commutative groups. This RAAG's contains both free groups, given by the complete graphs, and free abelian groups, given by disjoint vertices.

Warning

This is the opposite convention of some papers.

Right-angled Artin groups contain many remarkable properties and have a very rich structure despite their simple presentation. Here are some known facts:

- The word problem is solvable.
- They are known to be rigid; that is for any finite simple graphs Δ and Γ, we have $A_\Delta \cong A_\Gamma$ if and only if $\Delta \cong \Gamma$ [Dro1987].
- They embed as a finite index subgroup of a right-angled Coxeter group (which is the same definition as above except with the additional relations $g_v^2 = 1$ for all $v \in V(\Gamma)$).
- In [BB1997], it was shown they contain subgroups that satisfy the property FP_2 but are not finitely presented by considering the kernel of $\phi : A_\Gamma \to \mathbf{Z}$ by $g_v \mapsto 1$ (i.e. words of exponent sum 0).
- A_Γ has a finite $K(\pi, 1)$ space.
- A_Γ acts freely and cocompactly on a finite dimensional $CAT(0)$ space, and so it is biautomatic.
- Given an Artin group B with generators s_i, then any subgroup generated by a collection of $v_i = s_i^{k_i}$ where $k_i \geq 2$ is a RAAG where $[v_i, v_j] = 1$ if and only if $[s_i, s_j] = 1$ [CP2001].

The normal forms for RAAG's in Sage are those described in [VW1994] and gathers commuting groups together.

INPUT:

- `G` – a graph
- `names` – string or list of generator names

EXAMPLES:

```
sage: Gamma = Graph(4)
sage: G = RightAngledArtinGroup(Gamma)
sage: a,b,c,d = G.gens()
sage: a*c*d^4*a^-3*b
v0^-2*v1*v2*v3^4

sage: Gamma = graphs.CompleteGraph(4)
sage: G = RightAngledArtinGroup(Gamma)
sage: a,b,c,d = G.gens()
sage: a*c*d^4*a^-3*b
v0*v2*v3^4*v0^-3*v1

sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G
Right-angled Artin group of Cycle graph
sage: a,b,c,d,e = G.gens()
sage: d*b*a*d
v1*v3^2*v0
sage: e^-1*c*b*e*b^-1*c^-4
v2^-3
```

```
>>> from sage.all import *
>>> Gamma = Graph(Integer(4))
>>> G = RightAngledArtinGroup(Gamma)
>>> a,b,c,d = G.gens()
>>> a*c*d**Integer(4)*a**-Integer(3)*b
v0^-2*v1*v2*v3^4

>>> Gamma = graphs.CompleteGraph(Integer(4))
>>> G = RightAngledArtinGroup(Gamma)
>>> a,b,c,d = G.gens()
>>> a*c*d**Integer(4)*a**-Integer(3)*b
v0*v2*v3^4*v0^-3*v1

>>> Gamma = graphs.CycleGraph(Integer(5))
>>> G = RightAngledArtinGroup(Gamma)
>>> G
Right-angled Artin group of Cycle graph
>>> a,b,c,d,e = G.gens()
>>> d*b*a*d
v1*v3^2*v0
>>> e**-Integer(1)*c*b*e*b**-Integer(1)*c**-Integer(4)
v2^-3
```

We create the previous example but with different variable names:

```
sage: G.<a,b,c,d,e> = RightAngledArtinGroup(Gamma)
sage: G
Right-angled Artin group of Cycle graph
sage: d*b*a*d
b*d^2*a
sage: e^-1*c*b*e*b^-1*c^-4
c^-3
```

```
>>> from sage.all import *
>>> G = RightAngledArtinGroup(Gamma, names=('a', 'b', 'c', 'd', 'e',)); (a, b, c,
↪d, e,) = G._first_ngens(5)
>>> G
Right-angled Artin group of Cycle graph
>>> d*b*a*d
b*d^2*a
>>> e**-Integer(1)*c*b*e*b**-Integer(1)*c**-Integer(4)
c^-3
```

REFERENCES:

- [Cha2006]
- [BB1997]
- [Dro1987]
- [CP2001]
- [VW1994]
- Wikipedia article Artin_group#Right-angled_Artin_groups

class Element(*parent*, *lst*)

Bases: `ArtinGroupElement`

An element of a right-angled Artin group (RAAG).

Elements of RAAGs are modeled as lists of pairs `[i, p]` where `i` is the index of a vertex in the defining graph (with some fixed order of the vertices) and `p` is the power.

cohomology(*F=None*)

Return the cohomology ring of `self` over the field `F`.

EXAMPLES:

```
sage: C4 = graphs.CycleGraph(4)
sage: A = groups.misc.RightAngledArtin(C4)
sage: A.cohomology()
Cohomology ring of Right-angled Artin group of Cycle graph
 with coefficients in Rational Field
```

```
>>> from sage.all import *
>>> C4 = graphs.CycleGraph(Integer(4))
>>> A = groups.misc.RightAngledArtin(C4)
>>> A.cohomology()
Cohomology ring of Right-angled Artin group of Cycle graph
 with coefficients in Rational Field
```

gen(*i*)

Return the `i`-th generator of `self`.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.gen(2)
v2
```

```
>>> from sage.all import *
>>> Gamma = graphs.CycleGraph(Integer(5))
>>> G = RightAngledArtinGroup(Gamma)
>>> G.gen(Integer(2))
v2
```

gens()

Return the generators of `self`.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.gens()
(v0, v1, v2, v3, v4)
sage: Gamma = Graph([('x', 'y'), ('y', 'zeta')])
sage: G = RightAngledArtinGroup(Gamma)
sage: G.gens()
(vx, vy, vzeta)
```

```
>>> from sage.all import *
>>> Gamma = graphs.CycleGraph(Integer(5))
>>> G = RightAngledArtinGroup(Gamma)
>>> G.gens()
(v0, v1, v2, v3, v4)
>>> Gamma = Graph([('x', 'y'), ('y', 'zeta')])
>>> G = RightAngledArtinGroup(Gamma)
>>> G.gens()
(vx, vy, vzeta)
```

graph()

Return the defining graph of `self`.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.graph()
Cycle graph: Graph on 5 vertices
```

```
>>> from sage.all import *
>>> Gamma = graphs.CycleGraph(Integer(5))
>>> G = RightAngledArtinGroup(Gamma)
>>> G.graph()
Cycle graph: Graph on 5 vertices
```

ngens()

Return the number of generators of `self`.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.ngens()
5
```

```
>>> from sage.all import *
>>> Gamma = graphs.CycleGraph(Integer(5))
>>> G = RightAngledArtinGroup(Gamma)
>>> G.ngens()
5
```

one()

Return the identity element 1.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.one()
1
```

```
>>> from sage.all import *
>>> Gamma = graphs.CycleGraph(Integer(5))
>>> G = RightAngledArtinGroup(Gamma)
>>> G.one()
1
```

one_element()

Return the identity element 1.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.one()
1
```

```
>>> from sage.all import *
>>> Gamma = graphs.CycleGraph(Integer(5))
>>> G = RightAngledArtinGroup(Gamma)
>>> G.one()
1
```

CHAPTER

SEVENTEEN

CACTUS GROUPS

AUTHORS:

- Travis Scrimshaw (1-2023): initial version

class sage.groups.cactus_group.**CactusGroup**(*n*)

Bases: UniqueRepresentation, *Group*

The cactus group.

The n-fruit cactus group J_n is the group generated by s_{pq} for $1 \leq p < q \leq n$ with relations:

- $s_{pq}^2 = 1$
- $s_{pq}s_{kl} = s_{kl}s_{pq}$ if the intervals $[p, q]$ and $[k, l]$ are disjoint, and
- $s_{pq}s_{kl} = s_{p+q-l,p+q-k}s_{pq}$ if $[k, l] \subseteq [p, q]$.

INPUT:

- n – integer

EXAMPLES:

We construct the cactus group J_3 and do some basic computations:

```
sage: J3 = groups.misc.Cactus(3)
sage: s12,s13,s23 = J3.group_generators()
sage: s12 * s13
s[1,2]*s[1,3]
sage: x = s12 * s23; x
s[1,2]*s[2,3]
sage: x^4
s[1,2]*s[2,3]*s[1,2]*s[2,3]*s[1,2]*s[2,3]*s[1,2]*s[2,3]
sage: s12 * s13 == s13 * s23
True
```

```
>>> from sage.all import *
>>> J3 = groups.misc.Cactus(Integer(3))
>>> s12,s13,s23 = J3.group_generators()
>>> s12 * s13
s[1,2]*s[1,3]
>>> x = s12 * s23; x
s[1,2]*s[2,3]
>>> x**Integer(4)
s[1,2]*s[2,3]*s[1,2]*s[2,3]*s[1,2]*s[2,3]*s[1,2]*s[2,3]
```

(continues on next page)

(continued from previous page)

```
>>> s12 * s13 == s13 * s23
True
```

We verify the key equality in Lemma 2.3 in [White2015], which shows that J_5 is generated by s_{1q}:

```
sage: J5 = groups.misc.Cactus(5)
sage: gens = J5.group_generators()
sage: all(gens[(p,q)] == gens[(1,q)] * gens[(1,q-p+1)] * gens[(1,q)]
....:     for p in range(1, 6) for q in range(p+1, 6))
True
```

```
>>> from sage.all import *
>>> J5 = groups.misc.Cactus(Integer(5))
>>> gens = J5.group_generators()
>>> all(gens[(p,q)] == gens[(Integer(1),q)] * gens[(Integer(1),q-p+Integer(1))] *␣
↪gens[(Integer(1),q)]
...     for p in range(Integer(1), Integer(6)) for q in range(p+Integer(1),␣
↪Integer(6)))
True
```

class Element(*parent*, *data*)

Bases: `MultiplicativeGroupElement`

An element of a cactus group.

to_matrix()

Return `self` as a matrix.

The resulting matrix is the *geometric representation* of `self`.

EXAMPLES:

```
sage: J3 = groups.misc.Cactus(3)
sage: s12,s13,s23 = J3.gens()
sage: s12.to_matrix()
[ -1   0 2*t]
[  0   1   0]
[  0   0   1]
sage: (s12*s13).to_matrix()
[2*t   0  -1]
[  0  -1   0]
[  1   0   0]
sage: (s13*s23).to_matrix()
[2*t   0  -1]
[  0  -1   0]
[  1   0   0]
sage: (s13*s12).to_matrix()
[  0   0   1]
[  0  -1   0]
[ -1   0 2*t]
sage: all(x.to_matrix() * y.to_matrix() == (x*y).to_matrix()
....:     for x in J3.gens() for y in J3.gens())
True
```

```
>>> from sage.all import *
>>> J3 = groups.misc.Cactus(Integer(3))
>>> s12,s13,s23 = J3.gens()
>>> s12.to_matrix()
[ -1    0 2*t]
[  0    1    0]
[  0    0    1]
>>> (s12*s13).to_matrix()
[2*t    0   -1]
[  0   -1    0]
[  1    0    0]
>>> (s13*s23).to_matrix()
[2*t    0   -1]
[  0   -1    0]
[  1    0    0]
>>> (s13*s12).to_matrix()
[  0    0    1]
[  0   -1    0]
[ -1    0 2*t]
>>> all(x.to_matrix() * y.to_matrix() == (x*y).to_matrix()
...     for x in J3.gens() for y in J3.gens())
True
```

to_permutation()

Return the image of `self` under the canonical projection to the permutation group.

EXAMPLES:

```
sage: J3 = groups.misc.Cactus(3)
sage: s12,s13,s23 = J3.gens()
sage: s12.to_permutation()
[2, 1, 3]
sage: s23.to_permutation()
[1, 3, 2]
sage: s13.to_permutation()
[3, 2, 1]
sage: elt = s12*s23*s13
sage: elt.to_permutation()
[1, 3, 2]

sage: J7 = groups.misc.Cactus(7)
sage: J7.group_generators()[3,6].to_permutation()
[1, 2, 6, 5, 4, 3, 7]
```

```
>>> from sage.all import *
>>> J3 = groups.misc.Cactus(Integer(3))
>>> s12,s13,s23 = J3.gens()
>>> s12.to_permutation()
[2, 1, 3]
>>> s23.to_permutation()
[1, 3, 2]
>>> s13.to_permutation()
[3, 2, 1]
```

(continues on next page)

(continued from previous page)

```
>>> elt = s12*s23*s13
>>> elt.to_permutation()
[1, 3, 2]

>>> J7 = groups.misc.Cactus(Integer(7))
>>> J7.group_generators()[Integer(3),Integer(6)].to_permutation()
[1, 2, 6, 5, 4, 3, 7]
```

We check that this respects the multiplication order of permutations:

```
sage: P3 = Permutations(3)
sage: elt = s12*s23
sage: elt.to_permutation() == P3(s12) * P3(s23)
True
sage: Permutations.options.mult='r2l'
sage: elt.to_permutation() == P3(s12) * P3(s23)
True
sage: Permutations.options.mult='l2r'
```

```
>>> from sage.all import *
>>> P3 = Permutations(Integer(3))
>>> elt = s12*s23
>>> elt.to_permutation() == P3(s12) * P3(s23)
True
>>> Permutations.options.mult='r2l'
>>> elt.to_permutation() == P3(s12) * P3(s23)
True
>>> Permutations.options.mult='l2r'
```

bilinear_form(*t=None*)

Return the `t`-bilinear form of `self`.

We define a bilinear form B on the group algebra by

$$B(s_{ij}, s_{pq}) = \begin{cases} 1 & \text{if } i = p, j = q, \\ -t & \text{if } [i, j] \not\subseteq [p, q] \text{ and } [p, q] \not\subseteq [i, j], \\ 0 & \text{otherwise.} \end{cases}$$

In other words, it is 1 if $s_{ij} = s_{pq}$, $-t$ if s_{ij} and s_{pq} generate a free group, and 0 otherwise (they commute or almost commute).

INPUT:

- `t` – (default: t in $\mathbf{Z}[t]$) the variable t

EXAMPLES:

```
sage: J = groups.misc.Cactus(4)
sage: B = J.bilinear_form()
sage: B
[ 1  0  0 -t -t  0]
[ 0  1  0  0 -t -t]
[ 0  0  1  0  0  0]
[-t  0  0  1  0 -t]
```

(continues on next page)

(continued from previous page)

```
[-t -t  0  0  1  0]
[ 0 -t  0 -t  0  1]
```

```
>>> from sage.all import *
>>> J = groups.misc.Cactus(Integer(4))
>>> B = J.bilinear_form()
>>> B
[ 1  0  0 -t -t  0]
[ 0  1  0  0 -t -t]
[ 0  0  1  0  0  0]
[-t  0  0  1  0 -t]
[-t -t  0  0  1  0]
[ 0 -t  0 -t  0  1]
```

We reorder the generators so the bilinear form is more "Coxeter-like". In particular, when we remove the generator $s_{1,4}$, we recover the bilinear form in Example 6.2.5 of [DJS2003]:

```
sage: J.gens()
(s[1,2], s[1,3], s[1,4], s[2,3], s[2,4], s[3,4])
sage: S = SymmetricGroup(6)
sage: g = S([1,4,6,2,5,3])
sage: B.permute_rows_and_columns(g, g)
sage: B
[ 1 -t  0  0 -t  0]
[-t  1 -t  0  0  0]
[ 0 -t  1 -t  0  0]
[ 0  0 -t  1 -t  0]
[-t  0  0 -t  1  0]
[ 0  0  0  0  0  1]
```

```
>>> from sage.all import *
>>> J.gens()
(s[1,2], s[1,3], s[1,4], s[2,3], s[2,4], s[3,4])
>>> S = SymmetricGroup(Integer(6))
>>> g = S([Integer(1),Integer(4),Integer(6),Integer(2),Integer(5),Integer(3)])
>>> B.permute_rows_and_columns(g, g)
>>> B
[ 1 -t  0  0 -t  0]
[-t  1 -t  0  0  0]
[ 0 -t  1 -t  0  0]
[ 0  0 -t  1 -t  0]
[-t  0  0 -t  1  0]
[ 0  0  0  0  0  1]
```

gen (*i*, *j=None*)

Return the i-th generator of `self` or the generator s_{ij}.

EXAMPLES:

```
sage: J3 = groups.misc.Cactus(3)
sage: J3.gen(1)
s[1,3]
```

(continues on next page)

(continued from previous page)

```
sage: J3.gen(1,2)
s[1,2]
sage: J3.gen(0,2)
Traceback (most recent call last):
...
ValueError: s[0,2] is not a valid generator
sage: J3.gen(1,4)
Traceback (most recent call last):
...
ValueError: s[1,4] is not a valid generator
sage: J3.gen(2,1)
Traceback (most recent call last):
...
ValueError: s[2,1] is not a valid generator
```

```
>>> from sage.all import *
>>> J3 = groups.misc.Cactus(Integer(3))
>>> J3.gen(Integer(1))
s[1,3]
>>> J3.gen(Integer(1),Integer(2))
s[1,2]
>>> J3.gen(Integer(0),Integer(2))
Traceback (most recent call last):
...
ValueError: s[0,2] is not a valid generator
>>> J3.gen(Integer(1),Integer(4))
Traceback (most recent call last):
...
ValueError: s[1,4] is not a valid generator
>>> J3.gen(Integer(2),Integer(1))
Traceback (most recent call last):
...
ValueError: s[2,1] is not a valid generator
```

gens()

Return the generators of `self` as a tuple.

EXAMPLES:

```
sage: J3 = groups.misc.Cactus(3)
sage: J3.gens()
(s[1,2], s[1,3], s[2,3])
```

```
>>> from sage.all import *
>>> J3 = groups.misc.Cactus(Integer(3))
>>> J3.gens()
(s[1,2], s[1,3], s[2,3])
```

geometric_representation_generators(*t=None*)

Return the matrices corresponding to the generators of `self`.

We construct a representation over $R = \mathbf{Z}[t]$ of J_n as follows. Let E be the vector space over R spanned by $\{\epsilon_v\}_v$, where v is a generator of J_n. Fix some generator v, and let E_v denote the span of $\epsilon_u - \epsilon_{u'}$, where u'

is the reflected interval of u in v, over all u such that $u \subset v$. Let F_v denote the orthogonal complement of $R\epsilon_v \oplus E_v$ with respect to the `bilinear form` B. We define the action of v on E by

$$\rho(v) = -I|_{R\epsilon_v \oplus E_v} \oplus I|_{F_v}.$$

By Theorem 6.2.3 of [DJS2003], this defines a representation of J_n. It is expected that this is a faithful representation (see Remark 6.2.4 of [DJS2003]). As this arises from a blow-up and an analog of the geometric representation of the corresponding Coxeter group (the symmetric group), we call this the *geometric representation*.

INPUT:

- `t` – (default: t in $\mathbf{Z}[t]$) the variable t

EXAMPLES:

```
sage: J3 = groups.misc.Cactus(3)
sage: list(J3.geometric_representation_generators())
[
[ -1   0 2*t]  [ 0  0  1]  [  1   0   0]
[  0   1   0]  [ 0 -1  0]  [  0   1   0]
[  0   0   1], [ 1  0  0], [2*t   0  -1]
]
```

```
>>> from sage.all import *
>>> J3 = groups.misc.Cactus(Integer(3))
>>> list(J3.geometric_representation_generators())
[
[ -1   0 2*t]  [ 0  0  1]  [  1   0   0]
[  0   1   0]  [ 0 -1  0]  [  0   1   0]
[  0   0   1], [ 1  0  0], [2*t   0  -1]
]
```

We ran the following code with `max_tests = 15000` and did not find a counterexample to the faithfulness of this representation:

```
sage: visited = set([J3.one()])
sage: cur = set([(J3.one(), J3.one().to_matrix())])
sage: mats = set([J3.one().to_matrix()])
sage: RG = list(J3.geometric_representation_generators())
sage: count = 0
sage: max_tests = 1000
sage: while cur:
....:     count += 1
....:     if count >= max_tests:
....:         break
....:     elt, mat = cur.pop()
....:     for g,r in zip(J3, RG):
....:         val = elt * g
....:         if val in visited:
....:             continue
....:         visited.add(val)
....:         matp = mat * r
....:         matp.set_immutable()
....:         assert matp not in mats, f"not injective {val} \n{matp}"
```

(continues on next page)

(continued from previous page)

```
....:           mats.add(matp)
....:           cur.add((val, matp))
```

```
>>> from sage.all import *
>>> visited = set([J3.one()])
>>> cur = set([(J3.one(), J3.one().to_matrix())])
>>> mats = set([J3.one().to_matrix()])
>>> RG = list(J3.geometric_representation_generators())
>>> count = Integer(0)
>>> max_tests = Integer(1000)
>>> while cur:
...     count += Integer(1)
...     if count >= max_tests:
...         break
...     elt, mat = cur.pop()
...     for g,r in zip(J3, RG):
...         val = elt * g
...         if val in visited:
...             continue
...         visited.add(val)
...         matp = mat * r
...         matp.set_immutable()
...         assert matp not in mats, f"not injective {val} \n{matp}"
...         mats.add(matp)
...         cur.add((val, matp))
```

group_generators()

Return the group generators of `self`.

EXAMPLES:

```
sage: J3 = groups.misc.Cactus(3)
sage: J3.group_generators()
Finite family {(1, 2): s[1,2], (1, 3): s[1,3], (2, 3): s[2,3]}
```

```
>>> from sage.all import *
>>> J3 = groups.misc.Cactus(Integer(3))
>>> J3.group_generators()
Finite family {(1, 2): s[1,2], (1, 3): s[1,3], (2, 3): s[2,3]}
```

n()

Return the value n.

EXAMPLES:

```
sage: J3 = groups.misc.Cactus(3)
sage: J3.n()
3
```

```
>>> from sage.all import *
>>> J3 = groups.misc.Cactus(Integer(3))
>>> J3.n()
3
```

one()

Return the identity element in `self`.

EXAMPLES:

```
sage: J3 = groups.misc.Cactus(3)
sage: J3.one()
1
```

```
>>> from sage.all import *
>>> J3 = groups.misc.Cactus(Integer(3))
>>> J3.one()
1
```

random_element(*max_length=10*)

Return a random element of `self` of length at most `max_length`.

EXAMPLES:

```
sage: J3 = groups.misc.Cactus(3)
sage: J3.random_element()  # random
s[1,2]*s[2,3]*s[1,2]*s[1,3]
```

```
>>> from sage.all import *
>>> J3 = groups.misc.Cactus(Integer(3))
>>> J3.random_element()  # random
s[1,2]*s[2,3]*s[1,2]*s[1,3]
```

right_angled_coxeter_group()

Return the right-angled Coxeter group that `self` (set-theoretically) embeds into.

This is defined following [Most2019], where it was called the diagram group. It has generators (of order 2) indexed by subsets of $\{1, \ldots, n\}$ that commute if and only if one subset is contained in the other or they are disjoint. For the pure cactus group, this is also a group homomorphism, otherwise it is a group 1-cocycle [BCL2022].

EXAMPLES:

```
sage: J3 = groups.misc.Cactus(3)
sage: J3.right_angled_coxeter_group()
Coxeter group over Rational Field with Coxeter matrix:
[ 1 -1 -1  2]
[-1  1 -1  2]
[-1 -1  1  2]
[ 2  2  2  1]
```

```
>>> from sage.all import *
>>> J3 = groups.misc.Cactus(Integer(3))
>>> J3.right_angled_coxeter_group()
Coxeter group over Rational Field with Coxeter matrix:
[ 1 -1 -1  2]
[-1  1 -1  2]
[-1 -1  1  2]
[ 2  2  2  1]
```

class sage.groups.cactus_group.**PureCactusGroup**(*n*)

Bases: *KernelSubgroup*

The pure cactus group.

The *pure cactus group* PJ_n is the kernel of the natural surjection of the cactus group J_n onto the symmetric group S_n. In particular, we have the following (non-split) exact sequence:

$$1 \longrightarrow PJ_n \longrightarrow J_n \longrightarrow S_n \longrightarrow 1.$$

gen(*i*)

Return the i-th generator of self.

EXAMPLES:

```
sage: PJ3 = groups.misc.PureCactus(3)
sage: PJ3.gen(0)
s[2,3]*s[1,2]*s[2,3]*s[1,3]
sage: PJ3.gen(1)
s[1,2]*s[2,3]*s[1,2]*s[1,3]
sage: PJ3.gen(5)
Traceback (most recent call last):
...
IndexError: tuple index out of range
```

```
>>> from sage.all import *
>>> PJ3 = groups.misc.PureCactus(Integer(3))
>>> PJ3.gen(Integer(0))
s[2,3]*s[1,2]*s[2,3]*s[1,3]
>>> PJ3.gen(Integer(1))
s[1,2]*s[2,3]*s[1,2]*s[1,3]
>>> PJ3.gen(Integer(5))
Traceback (most recent call last):
...
IndexError: tuple index out of range
```

gens()

Return the generators of self.

ALGORITHM:

We use Wikipedia article Schreier's_lemma and compute the traversal using the lex minimum elements (defined by the order of the generators of the ambient cactus group).

EXAMPLES:

We verify Corollary A.2 of [BCL2022]:

```
sage: PJ3 = groups.misc.PureCactus(3)
sage: PJ3.gens()
(s[2,3]*s[1,2]*s[2,3]*s[1,3], s[1,2]*s[2,3]*s[1,2]*s[1,3])
sage: a, b = PJ3.gens()
sage: a * b  # they are inverses of each other
1
```

(continues on next page)

(continued from previous page)

```
sage: J3 = groups.misc.Cactus(3)
sage: gen = (J3.gen(1,2)*J3.gen(1,3))^3
sage: gen
s[1,2]*s[2,3]*s[1,2]*s[1,3]
sage: gen == b
True
```

```
>>> from sage.all import *
>>> PJ3 = groups.misc.PureCactus(Integer(3))
>>> PJ3.gens()
(s[2,3]*s[1,2]*s[2,3]*s[1,3], s[1,2]*s[2,3]*s[1,2]*s[1,3])
>>> a, b = PJ3.gens()
>>> a * b  # they are inverses of each other
1

>>> J3 = groups.misc.Cactus(Integer(3))
>>> gen = (J3.gen(Integer(1),Integer(2))*J3.gen(Integer(1),
↪Integer(3)))**Integer(3)
>>> gen
s[1,2]*s[2,3]*s[1,2]*s[1,3]
>>> gen == b
True
```

n()

Return the value n.

EXAMPLES:

```
sage: PJ3 = groups.misc.PureCactus(3)
sage: PJ3.n()
3
```

```
>>> from sage.all import *
>>> PJ3 = groups.misc.PureCactus(Integer(3))
>>> PJ3.n()
3
```

CHAPTER EIGHTEEN

FUNCTOR THAT CONVERTS A COMMUTATIVE ADDITIVE GROUP INTO A MULTIPLICATIVE GROUP.

AUTHORS:

- Mark Shimozono (2013): initial version

class sage.groups.group_exp.**GroupExp**

Bases: Functor

A functor that converts a commutative additive group into an isomorphic multiplicative group.

More precisely, given a commutative additive group G, define the exponential of G to be the isomorphic group with elements denoted e^g for every $g \in G$ and but with product in multiplicative notation

$$e^g e^h = e^{g+h} \qquad \text{for all } g, h \in G.$$

The class *GroupExp* implements the sage functor which sends a commutative additive group G to its exponential.

The creation of an instance of the functor *GroupExp* requires no input:

```
sage: E = GroupExp(); E
Functor from Category of commutative additive groups to Category of groups
```

```
>>> from sage.all import *
>>> E = GroupExp(); E
Functor from Category of commutative additive groups to Category of groups
```

The *GroupExp* functor (denoted E in the examples) can be applied to two kinds of input. The first is a commutative additive group. The output is its exponential. This is accomplished by `_apply_functor()`:

```
sage: EZ = E(ZZ); EZ
Multiplicative form of Integer Ring
```

```
>>> from sage.all import *
>>> EZ = E(ZZ); EZ
Multiplicative form of Integer Ring
```

Elements of the exponentiated group can be created and manipulated as follows:

```
sage: x = EZ(-3); x
-3
sage: x.parent()
Multiplicative form of Integer Ring
```

(continues on next page)

(continued from previous page)

```
sage: EZ(-1) * EZ(6) == EZ(5)
True
sage: EZ(3)^(-1)
-3
sage: EZ.one()
0
```

```
>>> from sage.all import *
>>> x = EZ(-Integer(3)); x
-3
>>> x.parent()
Multiplicative form of Integer Ring
>>> EZ(-Integer(1)) * EZ(Integer(6)) == EZ(Integer(5))
True
>>> EZ(Integer(3))**(-Integer(1))
-3
>>> EZ.one()
0
```

The second kind of input the `GroupExp` functor accepts, is a homomorphism of commutative additive groups. The output is the multiplicative form of the homomorphism. This is achieved by `_apply_functor_to_morphism()`:

```
sage: L = RootSystem(['A',2]).ambient_space()
sage: EL = E(L)
sage: W = L.weyl_group(prefix='s')
sage: s2 = W.simple_reflection(2)
sage: def my_action(mu):
....:     return s2.action(mu)
sage: from sage.categories.morphism import SetMorphism
sage: from sage.categories.homset import Hom
sage: f = SetMorphism(Hom(L, L, CommutativeAdditiveGroups()), my_action)
sage: F = E(f); F
Generic endomorphism of
 Multiplicative form of Ambient space of the Root system of type ['A', 2]
sage: v = L.an_element(); v
(2, 2, 3)
sage: y = F(EL(v)); y
(2, 3, 2)
sage: y.parent()
Multiplicative form of Ambient space of the Root system of type ['A', 2]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(2)]).ambient_space()
>>> EL = E(L)
>>> W = L.weyl_group(prefix='s')
>>> s2 = W.simple_reflection(Integer(2))
>>> def my_action(mu):
...     return s2.action(mu)
>>> from sage.categories.morphism import SetMorphism
>>> from sage.categories.homset import Hom
>>> f = SetMorphism(Hom(L, L, CommutativeAdditiveGroups()), my_action)
>>> F = E(f); F
```

(continues on next page)

(continued from previous page)

```
Generic endomorphism of
 Multiplicative form of Ambient space of the Root system of type ['A', 2]
>>> v = L.an_element(); v
(2, 2, 3)
>>> y = F(EL(v)); y
(2, 3, 2)
>>> y.parent()
Multiplicative form of Ambient space of the Root system of type ['A', 2]
```

class sage.groups.group_exp.**GroupExpElement**(*parent*, *x*)

Bases: ElementWrapper, MultiplicativeGroupElement

An element in the exponential of a commutative additive group.

INPUT:

- self – the exponentiated group element being created
- parent – the exponential group (parent of self)
- x – the commutative additive group element being wrapped to form self

EXAMPLES:

```
sage: from sage.groups.group_exp import GroupExpElement
sage: G = QQ^2
sage: EG = GroupExp()(G)
sage: z = GroupExpElement(EG, vector(QQ, (1, -3))); z
(1, -3)
sage: z.parent()
Multiplicative form of Vector space of dimension 2 over Rational Field
sage: EG(vector(QQ, (1, -3))) == z
True
```

```
>>> from sage.all import *
>>> from sage.groups.group_exp import GroupExpElement
>>> G = QQ**Integer(2)
>>> EG = GroupExp()(G)
>>> z = GroupExpElement(EG, vector(QQ, (Integer(1), -Integer(3)))); z
(1, -3)
>>> z.parent()
Multiplicative form of Vector space of dimension 2 over Rational Field
>>> EG(vector(QQ, (Integer(1), -Integer(3)))) == z
True
```

class sage.groups.group_exp.**GroupExp_Class**(G)

Bases: UniqueRepresentation, Parent

The multiplicative form of a commutative additive group.

INPUT:

- G – a commutative additive group

OUTPUT: the multiplicative form of G

EXAMPLES:

```
sage: GroupExp()(QQ)
Multiplicative form of Rational Field
```

```
>>> from sage.all import *
>>> GroupExp()(QQ)
Multiplicative form of Rational Field
```

Element

alias of *GroupExpElement*

an_element()

Return an element of the multiplicative group.

EXAMPLES:

```
sage: L = RootSystem(['A',2]).weight_lattice()
sage: EL = GroupExp()(L)
sage: x = EL.an_element(); x
2*Lambda[1] + 2*Lambda[2]
sage: x.parent()
Multiplicative form of Weight lattice of the Root system of type ['A', 2]
```

```
>>> from sage.all import *
>>> L = RootSystem(['A',Integer(2)]).weight_lattice()
>>> EL = GroupExp()(L)
>>> x = EL.an_element(); x
2*Lambda[1] + 2*Lambda[2]
>>> x.parent()
Multiplicative form of Weight lattice of the Root system of type ['A', 2]
```

group_generators()

Return generators of `self`.

EXAMPLES:

```
sage: GroupExp()(ZZ).group_generators()
(1,)
```

```
>>> from sage.all import *
>>> GroupExp()(ZZ).group_generators()
(1,)
```

one()

Return the identity element of the multiplicative group.

EXAMPLES:

```
sage: G = GroupExp()(ZZ^2)
sage: G.one()
(0, 0)
sage: x = G.an_element(); x
(1, 0)
sage: x == x * G.one()
True
```

```
>>> from sage.all import *
>>> G = GroupExp()(ZZ**Integer(2))
>>> G.one()
(0, 0)
>>> x = G.an_element(); x
(1, 0)
>>> x == x * G.one()
True
```

product (x, y)

Return the product of x and y in the multiplicative group.

EXAMPLES:

```
sage: G = GroupExp()(ZZ)
sage: G.product(G(2),G(7))
9
sage: x = G(2)
sage: x.__mul__(G(7))
9
```

```
>>> from sage.all import *
>>> G = GroupExp()(ZZ)
>>> G.product(G(Integer(2)),G(Integer(7)))
9
>>> x = G(Integer(2))
>>> x.__mul__(G(Integer(7)))
9
```

CHAPTER NINETEEN

SEMIDIRECT PRODUCT OF GROUPS

AUTHORS:

- Mark Shimozono (2013) initial version

class sage.groups.group_semidirect_product.**GroupSemidirectProduct**(*G*, *H*, *twist=None*, *act_to_right=True*, *prefix0=None*, *prefix1=None*, *print_tuple=False*, *category=Category of groups*)

Bases: `CartesianProduct`

Return the semidirect product of the groups `G` and `H` using the homomorphism `twist`.

INPUT:

- `G`, `H` – multiplicative groups
- `twist` – (default: `None`) a function defining a homomorphism (see below)
- `act_to_right` – boolean (default: `True`)
- `prefix0` – string (default: `None`)
- `prefix1` – string (default: `None`)
- `print_tuple` – boolean (default: `False`)
- `category` – a category (default: `Groups()`)

A semidirect product of groups G and H is a group structure on the Cartesian product $G \times H$ whose product agrees with that of G on $G \times 1_H$ and with that of H on $1_G \times H$, such that either $1_G \times H$ or $G \times 1_H$ is a normal subgroup. In the former case, the group is denoted $G \ltimes H$ and in the latter, $G \rtimes H$.

If `act_to_right` is `True`, this indicates the group $G \ltimes H$ in which G acts on H by automorphisms. In this case there is a group homomorphism $\phi \in \mathrm{Hom}(G, \mathrm{Aut}(H))$ such that

$$ghg^{-1} = \phi(g)(h).$$

The homomorphism ϕ is specified by the input `twist`, which syntactically is the function $G \times H \to H$ defined by

$$twist(g, h) = \phi(g)(h).$$

The product on $G \ltimes H$ is defined by

$$\begin{aligned}(g_1, h_1)(g_2, h_2) &= g_1 h_1 g_2 h_2 \\ &= g_1 g_2 g_2^{-1} h_1 g_2 h_2 \\ &= (g_1 g_2, twist(g_2^{-1}, h_1) h_2)\end{aligned}$$

If `act_to_right` is `False`, the group $G \rtimes H$ is specified by a homomorphism $\psi \in \mathrm{Hom}(H, \mathrm{Aut}(G))$ such that

$$hgh^{-1} = \psi(h)(g)$$

Then `twist` is the function $H \times G \to G$ defined by

$$twist(h, g) = \psi(h)(g).$$

so that the product in $G \rtimes H$ is defined by

$$\begin{aligned}(g_1, h_1)(g_2, h_2) &= g_1 h_1 g_2 h_2 \\ &= g_1 h_1 g_2 h_1^{-1} h_1 h_2 \\ &= (g_1 twist(h_1, g_2), h_1 h_2)\end{aligned}$$

If `prefix0` (resp. `prefixl`) is not `None` then it is used as a wrapper for printing elements of `G` (resp. `H`). If `print_tuple` is `True` then elements are printed in the style (g, h) and otherwise in the style $g * h$.

EXAMPLES:

```
sage: G = GL(2,QQ)
sage: V = QQ^2
sage: EV = GroupExp()(V)  # make a multiplicative version of V
sage: def twist(g, v):
....:     return EV(g*v.value)
sage: H = GroupSemidirectProduct(G, EV, twist=twist, prefix1='t'); H
Semidirect product of General Linear Group of degree 2
over Rational Field acting on Multiplicative form of Vector space
of dimension 2 over Rational Field
sage: x = H.an_element(); x
t[(1, 0)]
sage: x^2
t[(2, 0)]

sage: # needs sage.rings.number_field
sage: cartan_type = CartanType(['A',2])
sage: W = WeylGroup(cartan_type, prefix='s')
sage: def twist(w, v):
....:     return w*v*(~w)
sage: WW = GroupSemidirectProduct(W, W, twist=twist, print_tuple=True)
sage: s = Family(cartan_type.index_set(), lambda i: W.simple_reflection(i))
sage: y = WW((s[1],s[2])); y
(s1, s2)
sage: y^2
(1, s2*s1)
sage: y.inverse()
(s1, s1*s2*s1)
```

```
>>> from sage.all import *
>>> G = GL(Integer(2),QQ)
>>> V = QQ**Integer(2)
>>> EV = GroupExp()(V)  # make a multiplicative version of V
>>> def twist(g, v):
...     return EV(g*v.value)
>>> H = GroupSemidirectProduct(G, EV, twist=twist, prefix1='t'); H
```

(continues on next page)

(continued from previous page)

```
Semidirect product of General Linear Group of degree 2
over Rational Field acting on Multiplicative form of Vector space
of dimension 2 over Rational Field
>>> x = H.an_element(); x
t[(1, 0)]
>>> x**Integer(2)
t[(2, 0)]

>>> # needs sage.rings.number_field
>>> cartan_type = CartanType(['A',Integer(2)])
>>> W = WeylGroup(cartan_type, prefix='s')
>>> def twist(w, v):
...     return w*v*(~w)
>>> WW = GroupSemidirectProduct(W, W, twist=twist, print_tuple=True)
>>> s = Family(cartan_type.index_set(), lambda i: W.simple_reflection(i))
>>> y = WW((s[Integer(1)],s[Integer(2)])); y
(s1, s2)
>>> y**Integer(2)
(1, s2*s1)
>>> y.inverse()
(s1, s1*s2*s1)
```

Todo

- Functorial constructor for semidirect products for various categories
- Twofold Direct product as a special case of semidirect product

Element

alias of *GroupSemidirectProductElement*

act_to_right()

Return `True` if the left factor acts on the right factor and `False` if the right factor acts on the left factor.

EXAMPLES:

```
sage: def twist(x, y):
....:     return y
sage: GroupSemidirectProduct(WeylGroup(['A',2],prefix='s'),
....:                        WeylGroup(['A',3],prefix='t'), twist).act_to_
↪right()
True
```

```
>>> from sage.all import *
>>> def twist(x, y):
...     return y
>>> GroupSemidirectProduct(WeylGroup(['A',Integer(2)],prefix='s'),
...                        WeylGroup(['A',Integer(3)],prefix='t'), twist).act_
↪to_right()
True
```

construction()

Return None.

This overrides the construction functor inherited from CartesianProduct.

EXAMPLES:

```
sage: def twist(x, y):
....:     return y
sage: H = GroupSemidirectProduct(WeylGroup(['A',2],prefix='s'),
....:                            WeylGroup(['A',3],prefix='t'), twist)
sage: H.construction()
```

```
>>> from sage.all import *
>>> def twist(x, y):
...     return y
>>> H = GroupSemidirectProduct(WeylGroup(['A',Integer(2)],prefix='s'),
...                            WeylGroup(['A',Integer(3)],prefix='t'), twist)
>>> H.construction()
```

group_generators()

Return generators of self.

EXAMPLES:

```
sage: twist = lambda x,y: y
sage: import __main__
sage: __main__.twist = twist
sage: EZ = GroupExp()(ZZ)
sage: GroupSemidirectProduct(EZ, EZ, twist, print_tuple=True).group_
↪generators()
((1, 0), (0, 1))
```

```
>>> from sage.all import *
>>> twist = lambda x,y: y
>>> import __main__
>>> __main__.twist = twist
>>> EZ = GroupExp()(ZZ)
>>> GroupSemidirectProduct(EZ, EZ, twist, print_tuple=True).group_generators()
((1, 0), (0, 1))
```

one()

The identity element of the semidirect product group.

EXAMPLES:

```
sage: G = GL(2,QQ)
sage: V = QQ^2
sage: EV = GroupExp()(V)   # make a multiplicative version of V
sage: def twist(g, v):
....:     return EV(g*v.value)
sage: one = GroupSemidirectProduct(G, EV, twist=twist, prefix1='t').one(); one
1
sage: one.cartesian_projection(0)
[1 0]
```

(continues on next page)

(continued from previous page)

```
[0 1]
sage: one.cartesian_projection(1)
(0, 0)
```

```
>>> from sage.all import *
>>> G = GL(Integer(2),QQ)
>>> V = QQ**Integer(2)
>>> EV = GroupExp()(V)  # make a multiplicative version of V
>>> def twist(g, v):
...     return EV(g*v.value)
>>> one = GroupSemidirectProduct(G, EV, twist=twist, prefix1='t').one(); one
1
>>> one.cartesian_projection(Integer(0))
[1 0]
[0 1]
>>> one.cartesian_projection(Integer(1))
(0, 0)
```

opposite_semidirect_product()

Create the same semidirect product but with the positions of the groups exchanged.

EXAMPLES:

```
sage: G = GL(2,QQ)
sage: L = QQ^2
sage: EL = GroupExp()(L)
sage: H = GroupSemidirectProduct(G, EL, prefix1='t',
....:                            twist=lambda g,v: EL(g*v.value)); H
Semidirect product of General Linear Group of degree 2
over Rational Field acting on Multiplicative form of Vector space
of dimension 2 over Rational Field
sage: h = H((Matrix([[0,1],[1,0]]), EL.an_element())); h
[0 1]
[1 0] * t[(1, 0)]
sage: Hop = H.opposite_semidirect_product(); Hop
Semidirect product of Multiplicative form of Vector space
of dimension 2 over Rational Field acted upon by
General Linear Group of degree 2 over Rational Field
sage: hop = h.to_opposite(); hop
t[(0, 1)] * [0 1]
[1 0]
sage: hop in Hop
True
```

```
>>> from sage.all import *
>>> G = GL(Integer(2),QQ)
>>> L = QQ**Integer(2)
>>> EL = GroupExp()(L)
>>> H = GroupSemidirectProduct(G, EL, prefix1='t',
...                            twist=lambda g,v: EL(g*v.value)); H
Semidirect product of General Linear Group of degree 2
over Rational Field acting on Multiplicative form of Vector space
```

(continues on next page)

(continued from previous page)

```
of dimension 2 over Rational Field
>>> h = H((Matrix([[Integer(0),Integer(1)],[Integer(1),Integer(0)]]), EL.an_
→element())); h
[0 1]
[1 0] * t[(1, 0)]
>>> Hop = H.opposite_semidirect_product(); Hop
Semidirect product of Multiplicative form of Vector space
of dimension 2 over Rational Field acted upon by
General Linear Group of degree 2 over Rational Field
>>> hop = h.to_opposite(); hop
t[(0, 1)] * [0 1]
[1 0]
>>> hop in Hop
True
```

product (x, y)

The product of elements x and y in the semidirect product group.

EXAMPLES:

```
sage: G = GL(2,QQ)
sage: V = QQ^2
sage: EV = GroupExp()(V)  # make a multiplicative version of V
sage: def twist(g, v):
....:     return EV(g*v.value)
sage: S = GroupSemidirectProduct(G, EV, twist=twist, prefix1='t')
sage: g = G([[2,1],[3,1]]); g
[2 1]
[3 1]
sage: v = EV.an_element(); v
(1, 0)
sage: x = S((g,v)); x
[2 1]
[3 1] * t[(1, 0)]
sage: x*x # indirect doctest
[7 3]
[9 4] * t[(0, 3)]
```

```
>>> from sage.all import *
>>> G = GL(Integer(2),QQ)
>>> V = QQ**Integer(2)
>>> EV = GroupExp()(V)  # make a multiplicative version of V
>>> def twist(g, v):
...     return EV(g*v.value)
>>> S = GroupSemidirectProduct(G, EV, twist=twist, prefix1='t')
>>> g = G([[Integer(2),Integer(1)],[Integer(3),Integer(1)]]); g
[2 1]
[3 1]
>>> v = EV.an_element(); v
(1, 0)
>>> x = S((g,v)); x
[2 1]
```

(continues on next page)

(continued from previous page)

```
[3 1] * t[(1, 0)]
>>> x*x # indirect doctest
[7 3]
[9 4] * t[(0, 3)]
```

class sage.groups.group_semidirect_product.**GroupSemidirectProductElement**

Bases: Element

Element class for *GroupSemidirectProduct*.

to_opposite()

Send an element to its image in the opposite semidirect product.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: L = RootSystem(['A',2]).root_lattice(); L
Root lattice of the Root system of type ['A', 2]
sage: from sage.groups.group_exp import GroupExp
sage: EL = GroupExp()(L)
sage: W = L.weyl_group(prefix='s'); W
Weyl Group of type ['A', 2]
(as a matrix group acting on the root lattice)
sage: def twist(w, v):
....:     return EL(w.action(v.value))
sage: G = GroupSemidirectProduct(W, EL, twist, prefix1='t'); G
Semidirect product of Weyl Group of type ['A', 2] (as a matrix
group acting on the root lattice) acting on Multiplicative form of
Root lattice of the Root system of type ['A', 2]
sage: mu = L.an_element(); mu
2*alpha[1] + 2*alpha[2]
sage: w = W.an_element(); w
s1*s2
sage: g = G((w,EL(mu))); g
s1*s2 * t[2*alpha[1] + 2*alpha[2]]
sage: g.to_opposite()
t[-2*alpha[1]] * s1*s2
sage: g.to_opposite().parent()
Semidirect product of
 Multiplicative form of Root lattice of the Root system of type ['A', 2]
 acted upon by Weyl Group of type ['A', 2]
  (as a matrix group acting on the root lattice)
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> L = RootSystem(['A',Integer(2)]).root_lattice(); L
Root lattice of the Root system of type ['A', 2]
>>> from sage.groups.group_exp import GroupExp
>>> EL = GroupExp()(L)
>>> W = L.weyl_group(prefix='s'); W
Weyl Group of type ['A', 2]
(as a matrix group acting on the root lattice)
>>> def twist(w, v):
```

(continues on next page)

(continued from previous page)

```
...     return EL(w.action(v.value))
>>> G = GroupSemidirectProduct(W, EL, twist, prefix1='t'); G
Semidirect product of Weyl Group of type ['A', 2] (as a matrix
group acting on the root lattice) acting on Multiplicative form of
Root lattice of the Root system of type ['A', 2]
>>> mu = L.an_element(); mu
2*alpha[1] + 2*alpha[2]
>>> w = W.an_element(); w
s1*s2
>>> g = G((w,EL(mu))); g
s1*s2 * t[2*alpha[1] + 2*alpha[2]]
>>> g.to_opposite()
t[-2*alpha[1]] * s1*s2
>>> g.to_opposite().parent()
Semidirect product of
 Multiplicative form of Root lattice of the Root system of type ['A', 2]
 acted upon by Weyl Group of type ['A', 2]
  (as a matrix group acting on the root lattice)
```

CHAPTER

TWENTY

MISCELLANEOUS GROUPS

This is a collection of groups that may not fit into some of the other infinite families described elsewhere.

CHAPTER

TWENTYONE

SEMIMONOMIAL TRANSFORMATION GROUP

The semimonomial transformation group of degree n over a ring R is the semidirect product of the monomial transformation group of degree n (also known as the complete monomial group over the group of units $R^\times$ of R) and the group of ring automorphisms.

The multiplication of two elements $(\phi, \pi, \alpha)(\psi, \sigma, \beta)$ with

- $\phi, \psi \in {R^\times}^n$
- $\pi, \sigma \in S_n$ (with the multiplication $\pi\sigma$ done from left to right (like in GAP) – that is, $(\pi\sigma)(i) = \sigma(\pi(i))$ for all i.)
- $\alpha, \beta \in Aut(R)$

is defined by

$$(\phi, \pi, \alpha)(\psi, \sigma, \beta) = (\phi \cdot \psi^{\pi,\alpha}, \pi\sigma, \alpha \circ \beta)$$

where $\psi^{\pi,\alpha} = (\alpha(\psi_{\pi(1)-1}), \ldots, \alpha(\psi_{\pi(n)-1}))$ and the multiplication of vectors is defined elementwisely. (The indexing of vectors is 0-based here, so $\psi = (\psi_0, \psi_1, \ldots, \psi_{n-1})$.)

Todo

Up to now, this group is only implemented for finite fields because of the limited support of automorphisms for arbitrary rings.

AUTHORS:

- Thomas Feulner (2012-11-15): initial version

EXAMPLES:

```
sage: S = SemimonomialTransformationGroup(GF(4, 'a'), 4)
sage: G = S.gens()
sage: G[0]*G[1]
((a, 1, 1, 1); (1,2,3,4), Ring endomorphism of Finite Field in a of size 2^2
  Defn: a |--> a)
```

```
>>> from sage.all import *
>>> S = SemimonomialTransformationGroup(GF(Integer(4), 'a'), Integer(4))
>>> G = S.gens()
>>> G[Integer(0)]*G[Integer(1)]
((a, 1, 1, 1); (1,2,3,4), Ring endomorphism of Finite Field in a of size 2^2
  Defn: a |--> a)
```

class sage.groups.semimonomial_transformations.semimonomial_transformation_group.**SemimonomialActionMa**

Bases: Action

The left action of *SemimonomialTransformationGroup* on matrices over the same ring whose number of columns is equal to the degree. See *SemimonomialActionVec* for the definition of the action on the row vectors of such a matrix.

class sage.groups.semimonomial_transformations.semimonomial_transformation_group.**SemimonomialActionVe**

Bases: Action

The natural left action of the semimonomial group on vectors.

The action is defined by: $(\phi, \pi, \alpha) * (v_0, \ldots, v_{n-1}) := (\alpha(v_{\pi(1)-1}) \cdot \phi_0^{-1}, \ldots, \alpha(v_{\pi(n)-1}) \cdot \phi_{n-1}^{-1})$. (The indexing of vectors is 0-based here, so $\psi = (\psi_0, \psi_1, \ldots, \psi_{n-1})$.)

class sage.groups.semimonomial_transformations.semimonomial_transformation_group.**SemimonomialTransfor**

Bases: *FiniteGroup*, UniqueRepresentation

A semimonomial transformation group over a ring.

The semimonomial transformation group of degree n over a ring R is the semidirect product of the monomial transformation group of degree n (also known as the complete monomial group over the group of units $R^\times$ of R) and the group of ring automorphisms.

The multiplication of two elements $(\phi, \pi, \alpha)(\psi, \sigma, \beta)$ with

- $\phi, \psi \in R^{\times n}$
- $\pi, \sigma \in S_n$ (with the multiplication $\pi\sigma$ done from left to right (like in GAP) – that is, $(\pi\sigma)(i) = \sigma(\pi(i))$ for all i.)
- $\alpha, \beta \in Aut(R)$

is defined by

$$(\phi, \pi, \alpha)(\psi, \sigma, \beta) = (\phi \cdot \psi^{\pi,\alpha}, \pi\sigma, \alpha \circ \beta)$$

where $\psi^{\pi,\alpha} = (\alpha(\psi_{\pi(1)-1}), \ldots, \alpha(\psi_{\pi(n)-1}))$ and the multiplication of vectors is defined elementwisely. (The indexing of vectors is 0-based here, so $\psi = (\psi_0, \psi_1, \ldots, \psi_{n-1})$.)

Todo

Up to now, this group is only implemented for finite fields because of the limited support of automorphisms for arbitrary rings.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: S = SemimonomialTransformationGroup(F, 4)
sage: g = S(v = [2, a, 1, 2])
sage: h = S(perm = Permutation('(1,2,3,4)'), autom=F.hom([a**3]))
sage: g*h
((2, a, 1, 2); (1,2,3,4),
```

(continues on next page)

(continued from previous page)

```
 Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> 2*a + 1)
sage: h*g
((2*a + 1, 1, 2, 2); (1,2,3,4),
 Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> 2*a + 1)
sage: S(g)
((2, a, 1, 2); (),
 Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> a)
sage: S(1)
((1, 1, 1, 1); (),
 Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> a)
```

```
>>> from sage.all import *
>>> F = GF(Integer(9), names=('a',)); (a,) = F._first_ngens(1)
>>> S = SemimonomialTransformationGroup(F, Integer(4))
>>> g = S(v = [Integer(2), a, Integer(1), Integer(2)])
>>> h = S(perm = Permutation('(1,2,3,4)'), autom=F.hom([a**Integer(3)]))
>>> g*h
((2, a, 1, 2); (1,2,3,4),
 Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> 2*a + 1)
>>> h*g
((2*a + 1, 1, 2, 2); (1,2,3,4),
 Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> 2*a + 1)
>>> S(g)
((2, a, 1, 2); (),
 Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> a)
>>> S(Integer(1))
((1, 1, 1, 1); (),
 Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> a)
```

Element

alias of *SemimonomialTransformation*

base_ring()

Return the underlying ring of `self`.

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: SemimonomialTransformationGroup(F, 3).base_ring() is F
True
```

```
>>> from sage.all import *
>>> F = GF(Integer(4), names=('a',)); (a,) = F._first_ngens(1)
>>> SemimonomialTransformationGroup(F, Integer(3)).base_ring() is F
True
```

degree()

Return the degree of `self`.

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: SemimonomialTransformationGroup(F, 3).degree()
3
```

```
>>> from sage.all import *
>>> F = GF(Integer(4), names=('a',)); (a,) = F._first_ngens(1)
>>> SemimonomialTransformationGroup(F, Integer(3)).degree()
3
```

gens()

Return a tuple of generators of `self`.

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: SemimonomialTransformationGroup(F, 3).gens()
(((a, 1, 1); (),
  Ring endomorphism of Finite Field in a of size 2^2 Defn: a |--> a),
 ((1, 1, 1); (1,2,3),
  Ring endomorphism of Finite Field in a of size 2^2 Defn: a |--> a),
 ((1, 1, 1); (1,2),
  Ring endomorphism of Finite Field in a of size 2^2 Defn: a |--> a),
 ((1, 1, 1); (),
  Ring endomorphism of Finite Field in a of size 2^2 Defn: a |--> a + 1))
```

```
>>> from sage.all import *
>>> F = GF(Integer(4), names=('a',)); (a,) = F._first_ngens(1)
>>> SemimonomialTransformationGroup(F, Integer(3)).gens()
(((a, 1, 1); (),
  Ring endomorphism of Finite Field in a of size 2^2 Defn: a |--> a),
 ((1, 1, 1); (1,2,3),
  Ring endomorphism of Finite Field in a of size 2^2 Defn: a |--> a),
 ((1, 1, 1); (1,2),
  Ring endomorphism of Finite Field in a of size 2^2 Defn: a |--> a),
 ((1, 1, 1); (),
  Ring endomorphism of Finite Field in a of size 2^2 Defn: a |--> a + 1))
```

order()

Return the number of elements of `self`.

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: SemimonomialTransformationGroup(F, 5).order() == (4-1)**5 *␣
↪factorial(5) * 2
True
```

```
>>> from sage.all import *
>>> F = GF(Integer(4), names=('a',)); (a,) = F._first_ngens(1)
>>> SemimonomialTransformationGroup(F, Integer(5)).order() == (Integer(4)-
↪Integer(1))**Integer(5) * factorial(Integer(5)) * Integer(2)
True
```

CHAPTER

TWENTYTWO

ELEMENTS OF A SEMIMONOMIAL TRANSFORMATION GROUP

The semimonomial transformation group of degree n over a ring R is the semidirect product of the monomial transformation group of degree n (also known as the complete monomial group over the group of units $R^{\times}$ of R) and the group of ring automorphisms.

The multiplication of two elements $(\phi, \pi, \alpha)(\psi, \sigma, \beta)$ with

- $\phi, \psi \in {R^{\times}}^n$
- $\pi, \sigma \in S_n$ (with the multiplication $\pi\sigma$ done from left to right (like in GAP) – that is, $(\pi\sigma)(i) = \sigma(\pi(i))$ for all i.)
- $\alpha, \beta \in Aut(R)$

is defined by

$$(\phi, \pi, \alpha)(\psi, \sigma, \beta) = (\phi \cdot \psi^{\pi,\alpha}, \pi\sigma, \alpha \circ \beta)$$

with $\psi^{\pi,\alpha} = (\alpha(\psi_{\pi(1)-1}), \ldots, \alpha(\psi_{\pi(n)-1}))$ and an elementwisely defined multiplication of vectors. (The indexing of vectors is 0-based here, so $\psi = (\psi_0, \psi_1, \ldots, \psi_{n-1})$.)

The parent is `SemimonomialTransformationGroup`.

AUTHORS:

- Thomas Feulner (2012-11-15): initial version
- **Thomas Feulner (2013-12-27): Issue #15576 dissolve dependency on**
  Permutations.options.mul

EXAMPLES:

```
sage: S = SemimonomialTransformationGroup(GF(4, 'a'), 4)
sage: G = S.gens()
sage: G[0]*G[1]
((a, 1, 1, 1); (1,2,3,4), Ring endomorphism of Finite Field in a of size 2^2
  Defn: a |--> a)
```

```
>>> from sage.all import *
>>> S = SemimonomialTransformationGroup(GF(Integer(4), 'a'), Integer(4))
>>> G = S.gens()
>>> G[Integer(0)]*G[Integer(1)]
((a, 1, 1, 1); (1,2,3,4), Ring endomorphism of Finite Field in a of size 2^2
  Defn: a |--> a)
```

class sage.groups.semimonomial_transformations.semimonomial_transformation.
SemimonomialTransformation

Bases: `MultiplicativeGroupElement`

An element in the semimonomial group over a ring R. See *SemimonomialTransformationGroup* for the details on the multiplication of two elements.

The init method should never be called directly. Use the call via the parent *SemimonomialTransformation-Group*. instead.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: S = SemimonomialTransformationGroup(F, 4)
sage: g = S(v = [2, a, 1, 2])
sage: h = S(perm = Permutation('(1,2,3,4)'), autom=F.hom([a**3]))
sage: g*h
((2, a, 1, 2); (1,2,3,4), Ring endomorphism of Finite Field in a of size 3^2␣
↪Defn: a |--> 2*a + 1)
sage: h*g
((2*a + 1, 1, 2, 2); (1,2,3,4), Ring endomorphism of Finite Field in a of size 3^
↪2 Defn: a |--> 2*a + 1)
sage: S(g)
((2, a, 1, 2); (), Ring endomorphism of Finite Field in a of size 3^2 Defn: a |-->
↪ a)
sage: S(1)   # the one element in the group
((1, 1, 1, 1); (), Ring endomorphism of Finite Field in a of size 3^2 Defn: a |-->
↪ a)
```

```
>>> from sage.all import *
>>> F = GF(Integer(9), names=('a',)); (a,) = F._first_ngens(1)
>>> S = SemimonomialTransformationGroup(F, Integer(4))
>>> g = S(v = [Integer(2), a, Integer(1), Integer(2)])
>>> h = S(perm = Permutation('(1,2,3,4)'), autom=F.hom([a**Integer(3)]))
>>> g*h
((2, a, 1, 2); (1,2,3,4), Ring endomorphism of Finite Field in a of size 3^2␣
↪Defn: a |--> 2*a + 1)
>>> h*g
((2*a + 1, 1, 2, 2); (1,2,3,4), Ring endomorphism of Finite Field in a of size 3^
↪2 Defn: a |--> 2*a + 1)
>>> S(g)
((2, a, 1, 2); (), Ring endomorphism of Finite Field in a of size 3^2 Defn: a |-->
↪ a)
>>> S(Integer(1))   # the one element in the group
((1, 1, 1, 1); (), Ring endomorphism of Finite Field in a of size 3^2 Defn: a |-->
↪ a)
```

get_autom()

Return the component corresponding to $Aut(R)$ of `self`.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: SemimonomialTransformationGroup(F, 4).an_element().get_autom()
Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> 2*a + 1
```

```
>>> from sage.all import *
>>> F = GF(Integer(9), names=('a',)); (a,) = F._first_ngens(1)
>>> SemimonomialTransformationGroup(F, Integer(4)).an_element().get_autom()
Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> 2*a + 1
```

get_perm()

Return the component corresponding to S_n of `self`.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: SemimonomialTransformationGroup(F, 4).an_element().get_perm()
[4, 1, 2, 3]
```

```
>>> from sage.all import *
>>> F = GF(Integer(9), names=('a',)); (a,) = F._first_ngens(1)
>>> SemimonomialTransformationGroup(F, Integer(4)).an_element().get_perm()
[4, 1, 2, 3]
```

get_v()

Return the component corresponding to $R^{imes n}$ of `self`.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: SemimonomialTransformationGroup(F, 4).an_element().get_v()
(a, 1, 1, 1)
```

```
>>> from sage.all import *
>>> F = GF(Integer(9), names=('a',)); (a,) = F._first_ngens(1)
>>> SemimonomialTransformationGroup(F, Integer(4)).an_element().get_v()
(a, 1, 1, 1)
```

get_v_inverse()

Return the (elementwise) inverse of the component corresponding to $R^{imes n}$ of `self`.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: SemimonomialTransformationGroup(F, 4).an_element().get_v_inverse()
(a + 2, 1, 1, 1)
```

```
>>> from sage.all import *
>>> F = GF(Integer(9), names=('a',)); (a,) = F._first_ngens(1)
>>> SemimonomialTransformationGroup(F, Integer(4)).an_element().get_v_
↪inverse()
(a + 2, 1, 1, 1)
```

invert_v()

Elementwisely invert all entries of `self` which correspond to the component $R^{imes n}$.

The other components of `self` keep unchanged.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: x = copy(SemimonomialTransformationGroup(F, 4).an_element())
sage: x.invert_v()
sage: x.get_v() == SemimonomialTransformationGroup(F, 4).an_element().get_v_
→inverse()
True
```

```
>>> from sage.all import *
>>> F = GF(Integer(9), names=('a',)); (a,) = F._first_ngens(1)
>>> x = copy(SemimonomialTransformationGroup(F, Integer(4)).an_element())
>>> x.invert_v()
>>> x.get_v() == SemimonomialTransformationGroup(F, Integer(4)).an_element().
→get_v_inverse()
True
```

CHAPTER

TWENTYTHREE

KERNEL SUBGROUPS

The kernel of a homomorphism implemented as a subgroup.

AUTHORS:

- Travis Scrimshaw (1-2023): Initial version

class sage.groups.kernel_subgroup.**KernelSubgroup**(*morphism*)

Bases: UniqueRepresentation, Parent

The kernel (normal) subgroup.

Let $\phi : G \to H$ be a group homomorphism. The kernel $K = \{\phi(g) = 1 | g \in G\}$ is a normal subgroup of G.

class Element

Bases: ElementWrapper

ambient()

Return the ambient group of self.

EXAMPLES:

```
sage: PJ3 = groups.misc.PureCactus(3)                                    #␣
↪needs sage.rings.number_field
sage: PJ3.ambient()                                                      #␣
↪needs sage.rings.number_field
Cactus Group with 3 fruit
```

```
>>> from sage.all import *
>>> PJ3 = groups.misc.PureCactus(Integer(3))                               ␣
↪     # needs sage.rings.number_field
>>> PJ3.ambient()                                                       #␣
↪needs sage.rings.number_field
Cactus Group with 3 fruit
```

defining_morphism()

Return the defining morphism of self.

EXAMPLES:

```
sage: PJ3 = groups.misc.PureCactus(3)                                    #␣
↪needs sage.rings.number_field
sage: PJ3.defining_morphism()                                            #␣
↪needs sage.rings.number_field
Conversion via _from_cactus_group_element map:
```

(continues on next page)

(continued from previous page)

```
  From: Cactus Group with 3 fruit
  To:   Symmetric group of order 3! as a permutation group
```

```
>>> from sage.all import *
>>> PJ3 = groups.misc.PureCactus(Integer(3))                                ␣
↪     # needs sage.rings.number_field
>>> PJ3.defining_morphism()                                                #␣
↪needs sage.rings.number_field
Conversion via _from_cactus_group_element map:
  From: Cactus Group with 3 fruit
  To:   Symmetric group of order 3! as a permutation group
```

gens()

Return the generators of `self`.

EXAMPLES:

```
sage: S2 = SymmetricGroup(2)
sage: S3 = SymmetricGroup(3)
sage: H = Hom(S3, S2)
sage: phi = H(S2.__call__)
sage: from sage.groups.kernel_subgroup import KernelSubgroup
sage: K = KernelSubgroup(phi)
sage: K.gens()
((),)
```

```
>>> from sage.all import *
>>> S2 = SymmetricGroup(Integer(2))
>>> S3 = SymmetricGroup(Integer(3))
>>> H = Hom(S3, S2)
>>> phi = H(S2.__call__)
>>> from sage.groups.kernel_subgroup import KernelSubgroup
>>> K = KernelSubgroup(phi)
>>> K.gens()
((),)
```

lift(*x*)

Lift `x` to the ambient group of `self`.

EXAMPLES:

```
sage: PJ3 = groups.misc.PureCactus(3)                                       #␣
↪needs sage.rings.number_field
sage: PJ3.lift(PJ3.an_element()).parent()                                   #␣
↪needs sage.rings.number_field
Cactus Group with 3 fruit
```

```
>>> from sage.all import *
>>> PJ3 = groups.misc.PureCactus(Integer(3))                                ␣
↪     # needs sage.rings.number_field
>>> PJ3.lift(PJ3.an_element()).parent()                                    #␣
↪needs sage.rings.number_field
Cactus Group with 3 fruit
```

retract (*x*)

Convert `x` to an element of `self`.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: J3 = groups.misc.Cactus(3)
sage: s12,s13,s23 = J3.group_generators()
sage: PJ3 = groups.misc.PureCactus(3)
sage: elt = PJ3.retract(s23*s12*s23*s13); elt
s[2,3]*s[1,2]*s[2,3]*s[1,3]
sage: elt.parent() is PJ3
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> J3 = groups.misc.Cactus(Integer(3))
>>> s12,s13,s23 = J3.group_generators()
>>> PJ3 = groups.misc.PureCactus(Integer(3))
>>> elt = PJ3.retract(s23*s12*s23*s13); elt
s[2,3]*s[1,2]*s[2,3]*s[1,3]
>>> elt.parent() is PJ3
True
```

CHAPTER

TWENTYFOUR

CLASS FUNCTIONS OF GROUPS.

This module implements a wrapper of GAP's ClassFunction function.

Note

The ordering of the columns of the character table of a group corresponds to the ordering of the list. However, in general there is no way to canonically list (or index) the conjugacy classes of a group. Therefore the ordering of the columns of the character table of a group is somewhat random.

AUTHORS:

- Franco Saliola (November 2008): initial version
- Volker Braun (October 2010): Bugfixes, exterior and symmetric power.

`sage.groups.class_function.`**`ClassFunction`**`(`*group*, *values*`)`

Construct a class function.

INPUT:

- `group` – a group
- `values` – list/tuple/iterable of numbers; the values of the class function on the conjugacy classes, in that order

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: G.conjugacy_classes()
[Conjugacy class of () in Cyclic group of order 4 as a permutation group,
 Conjugacy class of (1,2,3,4) in Cyclic group of order 4 as a permutation group,
 Conjugacy class of (1,3)(2,4) in Cyclic group of order 4 as a permutation group,
 Conjugacy class of (1,4,3,2) in Cyclic group of order 4 as a permutation group]
sage: values  = [1, -1, 1, -1]
sage: chi = ClassFunction(G, values); chi
Character of Cyclic group of order 4 as a permutation group
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(4))
>>> G.conjugacy_classes()
[Conjugacy class of () in Cyclic group of order 4 as a permutation group,
 Conjugacy class of (1,2,3,4) in Cyclic group of order 4 as a permutation group,
 Conjugacy class of (1,3)(2,4) in Cyclic group of order 4 as a permutation group,
 Conjugacy class of (1,4,3,2) in Cyclic group of order 4 as a permutation group]
```

(continues on next page)

(continued from previous page)

```
>>> values  = [Integer(1), -Integer(1), Integer(1), -Integer(1)]
>>> chi = ClassFunction(G, values); chi
Character of Cyclic group of order 4 as a permutation group
```

class sage.groups.class_function.**ClassFunction_gap**(*G*, *values*)

Bases: SageObject

A wrapper of GAP's ClassFunction function.

> **Note**
>
> It is *not* checked whether the given values describes a character, since GAP does not do this.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: values  = [1, -1, 1, -1]
sage: chi = ClassFunction(G, values); chi
Character of Cyclic group of order 4 as a permutation group
sage: loads(dumps(chi)) == chi
True
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(4))
>>> values  = [Integer(1), -Integer(1), Integer(1), -Integer(1)]
>>> chi = ClassFunction(G, values); chi
Character of Cyclic group of order 4 as a permutation group
>>> loads(dumps(chi)) == chi
True
```

adams_operation(*k*)

Return the k-th Adams operation on self.

Let G be a finite group. The k-th Adams operation Ψ^k is given by

$$\Psi^k(\chi)(g) = \chi(g^k).$$

The Adams operations turn the representation ring of G into a λ-ring.

EXAMPLES:

```
sage: G = groups.permutation.Alternating(5)
sage: chars = G.irreducible_characters()
sage: [chi.adams_operation(2).values() for chi in chars]
[[1, 1, 1, 1, 1],
 [3, 3, 0, -zeta5^3 - zeta5^2, zeta5^3 + zeta5^2 + 1],
 [3, 3, 0, zeta5^3 + zeta5^2 + 1, -zeta5^3 - zeta5^2],
 [4, 4, 1, -1, -1],
 [5, 5, -1, 0, 0]]
sage: chars[4].adams_operation(2).decompose()
((1, Character of Alternating group of order 5!/2 as a permutation group),
 (-1, Character of Alternating group of order 5!/2 as a permutation group),
```

(continues on next page)

(continued from previous page)

```
 (-1, Character of Alternating group of order 5!/2 as a permutation group),
 (2, Character of Alternating group of order 5!/2 as a permutation group))
```

```
>>> from sage.all import *
>>> G = groups.permutation.Alternating(Integer(5))
>>> chars = G.irreducible_characters()
>>> [chi.adams_operation(Integer(2)).values() for chi in chars]
[[1, 1, 1, 1, 1],
 [3, 3, 0, -zeta5^3 - zeta5^2, zeta5^3 + zeta5^2 + 1],
 [3, 3, 0, zeta5^3 + zeta5^2 + 1, -zeta5^3 - zeta5^2],
 [4, 4, 1, -1, -1],
 [5, 5, -1, 0, 0]]
>>> chars[Integer(4)].adams_operation(Integer(2)).decompose()
((1, Character of Alternating group of order 5!/2 as a permutation group),
 (-1, Character of Alternating group of order 5!/2 as a permutation group),
 (-1, Character of Alternating group of order 5!/2 as a permutation group),
 (2, Character of Alternating group of order 5!/2 as a permutation group))
```

REFERENCES:

- Wikipedia article Adams_operation

central_character()

Return the central character of `self`.

EXAMPLES:

```
sage: t = SymmetricGroup(4).trivial_character()
sage: t.central_character().values()
[1, 6, 3, 8, 6]
```

```
>>> from sage.all import *
>>> t = SymmetricGroup(Integer(4)).trivial_character()
>>> t.central_character().values()
[1, 6, 3, 8, 6]
```

decompose()

Return a list of the characters appearing the decomposition of `self`.

EXAMPLES:

```
sage: S5 = SymmetricGroup(5)
sage: chi = ClassFunction(S5, [22, -8, 2, 1, 1, 2, -3])
sage: chi.decompose()
((3, Character of Symmetric group of order 5! as a permutation group),
 (2, Character of Symmetric group of order 5! as a permutation group))
```

```
>>> from sage.all import *
>>> S5 = SymmetricGroup(Integer(5))
>>> chi = ClassFunction(S5, [Integer(22), -Integer(8), Integer(2), Integer(1),
↪ Integer(1), Integer(2), -Integer(3)])
>>> chi.decompose()
((3, Character of Symmetric group of order 5! as a permutation group),
 (2, Character of Symmetric group of order 5! as a permutation group))
```

degree()

Return the degree of the character `self`.

EXAMPLES:

```
sage: S5 = SymmetricGroup(5)
sage: irr = S5.irreducible_characters()
sage: [x.degree() for x in irr]
[1, 4, 5, 6, 5, 4, 1]
```

```
>>> from sage.all import *
>>> S5 = SymmetricGroup(Integer(5))
>>> irr = S5.irreducible_characters()
>>> [x.degree() for x in irr]
[1, 4, 5, 6, 5, 4, 1]
```

determinant_character()

Return the determinant character of `self`.

EXAMPLES:

```
sage: t = ClassFunction(SymmetricGroup(4), [1, -1, 1, 1, -1])
sage: t.determinant_character().values()
[1, -1, 1, 1, -1]
```

```
>>> from sage.all import *
>>> t = ClassFunction(SymmetricGroup(Integer(4)), [Integer(1), -Integer(1),␣
↪Integer(1), Integer(1), -Integer(1)])
>>> t.determinant_character().values()
[1, -1, 1, 1, -1]
```

domain()

Return the domain of the `self`.

OUTPUT: the underlying group of the class function

EXAMPLES:

```
sage: ClassFunction(SymmetricGroup(4), [1,-1,1,1,-1]).domain()
Symmetric group of order 4! as a permutation group
```

```
>>> from sage.all import *
>>> ClassFunction(SymmetricGroup(Integer(4)), [Integer(1),-Integer(1),
↪Integer(1),Integer(1),-Integer(1)]).domain()
Symmetric group of order 4! as a permutation group
```

exterior_power(n)

Return the antisymmetrized product of `self` with itself n times.

INPUT:

- `n` – positive integer

OUTPUT: the n-th antisymmetrized power of `self` as a `ClassFunction`

EXAMPLES:

```
sage: chi = ClassFunction(SymmetricGroup(4), gap([3, 1, -1, 0, -1]))
sage: p = chi.exterior_power(3)    # the highest antisymmetric power for a 3-d␣
↪character
sage: p
Character of Symmetric group of order 4! as a permutation group
sage: p.values()
[1, -1, 1, 1, -1]
sage: p == chi.determinant_character()
True
```

```
>>> from sage.all import *
>>> chi = ClassFunction(SymmetricGroup(Integer(4)), gap([Integer(3),␣
↪Integer(1), -Integer(1), Integer(0), -Integer(1)]))
>>> p = chi.exterior_power(Integer(3))    # the highest antisymmetric power␣
↪for a 3-d character
>>> p
Character of Symmetric group of order 4! as a permutation group
>>> p.values()
[1, -1, 1, 1, -1]
>>> p == chi.determinant_character()
True
```

induct(*G*)

Return the induced character.

INPUT:

- `G` – a supergroup of the underlying group of `self`

OUTPUT:

A *ClassFunction* of `G` defined by induction. Induction is the adjoint functor to restriction, see *restrict()*.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: H = G.subgroup([(1,2,3), (1,2), (4,5)])
sage: xi = H.trivial_character(); xi
Character of Subgroup generated by [(1,2,3), (1,2), (4,5)] of
 (Symmetric group of order 5! as a permutation group)
sage: xi.induct(G)
Character of Symmetric group of order 5! as a permutation group
sage: xi.induct(G).values()
[10, 4, 2, 1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> H = G.subgroup([(Integer(1),Integer(2),Integer(3)), (Integer(1),
↪Integer(2)), (Integer(4),Integer(5))])
>>> xi = H.trivial_character(); xi
Character of Subgroup generated by [(1,2,3), (1,2), (4,5)] of
 (Symmetric group of order 5! as a permutation group)
>>> xi.induct(G)
Character of Symmetric group of order 5! as a permutation group
```

(continues on next page)

(continued from previous page)

```
>>> xi.induct(G).values()
[10, 4, 2, 1, 1, 0, 0]
```

irreducible_constituents()

Return a list of the characters that appear in the decomposition of chi.

EXAMPLES:

```
sage: S5 = SymmetricGroup(5)
sage: chi = ClassFunction(S5, [22, -8, 2, 1, 1, 2, -3])
sage: irr = chi.irreducible_constituents(); irr
(Character of Symmetric group of order 5! as a permutation group,
 Character of Symmetric group of order 5! as a permutation group)
sage: list(map(list, irr))
[[4, -2, 0, 1, 1, 0, -1], [5, -1, 1, -1, -1, 1, 0]]
sage: G = GL(2,3)
sage: chi = ClassFunction(G, [-1, -1, -1, -1, -1, -1, -1, -1])
sage: chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
sage: chi = ClassFunction(G, [1, 1, 1, 1, 1, 1, 1, 1])
sage: chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
sage: chi = ClassFunction(G, [2, 2, 2, 2, 2, 2, 2, 2])
sage: chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
sage: chi = ClassFunction(G, [-1, -1, -1, -1, 3, -1, -1, 1])
sage: ic = chi.irreducible_constituents(); ic
(Character of General Linear Group of degree 2 over Finite Field of size 3,
 Character of General Linear Group of degree 2 over Finite Field of size 3)
sage: list(map(list, ic))
[[2, -1, 2, -1, 2, 0, 0, 0], [3, 0, 3, 0, -1, 1, 1, -1]]
```

```
>>> from sage.all import *
>>> S5 = SymmetricGroup(Integer(5))
>>> chi = ClassFunction(S5, [Integer(22), -Integer(8), Integer(2), Integer(1),
↪ Integer(1), Integer(2), -Integer(3)])
>>> irr = chi.irreducible_constituents(); irr
(Character of Symmetric group of order 5! as a permutation group,
 Character of Symmetric group of order 5! as a permutation group)
>>> list(map(list, irr))
[[4, -2, 0, 1, 1, 0, -1], [5, -1, 1, -1, -1, 1, 0]]
>>> G = GL(Integer(2),Integer(3))
>>> chi = ClassFunction(G, [-Integer(1), -Integer(1), -Integer(1), -
↪Integer(1), -Integer(1), -Integer(1), -Integer(1), -Integer(1)])
>>> chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
>>> chi = ClassFunction(G, [Integer(1), Integer(1), Integer(1), Integer(1),␣
↪Integer(1), Integer(1), Integer(1), Integer(1)])
>>> chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
>>> chi = ClassFunction(G, [Integer(2), Integer(2), Integer(2), Integer(2),␣
↪Integer(2), Integer(2), Integer(2), Integer(2)])
```

(continues on next page)

(continued from previous page)

```
>>> chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
>>> chi = ClassFunction(G, [-Integer(1), -Integer(1), -Integer(1), -
↪Integer(1), Integer(3), -Integer(1), -Integer(1), Integer(1)])
>>> ic = chi.irreducible_constituents(); ic
(Character of General Linear Group of degree 2 over Finite Field of size 3,
 Character of General Linear Group of degree 2 over Finite Field of size 3)
>>> list(map(list, ic))
[[2, -1, 2, -1, 2, 0, 0, 0], [3, 0, 3, 0, -1, 1, 1, -1]]
```

is_irreducible()

Return `True` if `self` cannot be written as the sum of two nonzero characters of `self`.

EXAMPLES:

```
sage: S4 = SymmetricGroup(4)
sage: irr = S4.irreducible_characters()
sage: [x.is_irreducible() for x in irr]
[True, True, True, True, True]
```

```
>>> from sage.all import *
>>> S4 = SymmetricGroup(Integer(4))
>>> irr = S4.irreducible_characters()
>>> [x.is_irreducible() for x in irr]
[True, True, True, True, True]
```

norm()

Return the norm of `self`.

EXAMPLES:

```
sage: A5 = AlternatingGroup(5)
sage: [x.norm() for x in A5.irreducible_characters()]
[1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> A5 = AlternatingGroup(Integer(5))
>>> [x.norm() for x in A5.irreducible_characters()]
[1, 1, 1, 1, 1]
```

restrict(*H*)

Return the restricted character.

INPUT:

- `H` – a subgroup of the underlying group of `self`

OUTPUT: a *ClassFunction* of `H` defined by restriction

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: chi = ClassFunction(G, [3, -3, -1, 0, 0, -1, 3]); chi
Character of Symmetric group of order 5! as a permutation group
```

(continues on next page)

(continued from previous page)

```
sage: H = G.subgroup([(1,2,3), (1,2), (4,5)])
sage: chi.restrict(H)
Character of Subgroup generated by [(1,2,3), (1,2), (4,5)] of
 (Symmetric group of order 5! as a permutation group)
sage: chi.restrict(H).values()
[3, -3, -3, -1, 0, 0]
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> chi = ClassFunction(G, [Integer(3), -Integer(3), -Integer(1), Integer(0),␣
↪Integer(0), -Integer(1), Integer(3)]); chi
Character of Symmetric group of order 5! as a permutation group
>>> H = G.subgroup([(Integer(1),Integer(2),Integer(3)), (Integer(1),
↪Integer(2)), (Integer(4),Integer(5))])
>>> chi.restrict(H)
Character of Subgroup generated by [(1,2,3), (1,2), (4,5)] of
 (Symmetric group of order 5! as a permutation group)
>>> chi.restrict(H).values()
[3, -3, -3, -1, 0, 0]
```

scalar_product(*other*)

Return the scalar product of `self` with `other`.

EXAMPLES:

```
sage: S4 = SymmetricGroup(4)
sage: irr = S4.irreducible_characters()
sage: [[x.scalar_product(y) for x in irr] for y in irr]
[[1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 1]]
```

```
>>> from sage.all import *
>>> S4 = SymmetricGroup(Integer(4))
>>> irr = S4.irreducible_characters()
>>> [[x.scalar_product(y) for x in irr] for y in irr]
[[1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 1]]
```

symmetric_power(*n*)

Return the symmetrized product of `self` with itself `n` times.

INPUT:

- `n` – positive integer

OUTPUT: the `n`-th symmetrized power of `self` as a *ClassFunction*

EXAMPLES:

```
sage: chi = ClassFunction(SymmetricGroup(4), gap([3, 1, -1, 0, -1]))
sage: p = chi.symmetric_power(3)
sage: p
Character of Symmetric group of order 4! as a permutation group
sage: p.values()
[10, 2, -2, 1, 0]
```

```
>>> from sage.all import *
>>> chi = ClassFunction(SymmetricGroup(Integer(4)), gap([Integer(3),␣
↪Integer(1), -Integer(1), Integer(0), -Integer(1)]))
>>> p = chi.symmetric_power(Integer(3))
>>> p
Character of Symmetric group of order 4! as a permutation group
>>> p.values()
[10, 2, -2, 1, 0]
```

tensor_product(*other*)

EXAMPLES:

```
sage: S3 = SymmetricGroup(3)
sage: chi1, chi2, chi3 = S3.irreducible_characters()
sage: chi1.tensor_product(chi3).values()
[1, -1, 1]
```

```
>>> from sage.all import *
>>> S3 = SymmetricGroup(Integer(3))
>>> chi1, chi2, chi3 = S3.irreducible_characters()
>>> chi1.tensor_product(chi3).values()
[1, -1, 1]
```

values()

Return the list of values of `self` on the conjugacy classes.

EXAMPLES:

```
sage: G = GL(2,3)
sage: [x.values() for x in G.irreducible_characters()] #random
[[1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, -1, -1, -1],
 [2, -1, 2, -1, 2, 0, 0, 0],
 [2, 1, -2, -1, 0, -zeta8^3 - zeta8, zeta8^3 + zeta8, 0],
 [2, 1, -2, -1, 0, zeta8^3 + zeta8, -zeta8^3 - zeta8, 0],
 [3, 0, 3, 0, -1, -1, -1, 1],
 [3, 0, 3, 0, -1, 1, 1, -1],
 [4, -1, -4, 1, 0, 0, 0, 0]]
```

```
>>> from sage.all import *
>>> G = GL(Integer(2),Integer(3))
>>> [x.values() for x in G.irreducible_characters()] #random
[[1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, -1, -1, -1],
 [2, -1, 2, -1, 2, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
[2, 1, -2, -1, 0, -zeta8^3 - zeta8, zeta8^3 + zeta8, 0],
[2, 1, -2, -1, 0, zeta8^3 + zeta8, -zeta8^3 - zeta8, 0],
[3, 0, 3, 0, -1, -1, -1, 1],
[3, 0, 3, 0, -1, 1, 1, -1],
[4, -1, -4, 1, 0, 0, 0, 0]]
```

class sage.groups.class_function.**ClassFunction_libgap**(*G*, *values*)

Bases: `SageObject`

A wrapper of GAP's `ClassFunction` function.

Note

It is *not* checked whether the given values describes a character, since GAP does not do this.

EXAMPLES:

```
sage: G = SO(3,3)
sage: values  = [1, -1, -1, 1, 2]
sage: chi = ClassFunction(G, values); chi
Character of Special Orthogonal Group of degree 3 over Finite Field of size 3
sage: loads(dumps(chi)) == chi
True
```

```
>>> from sage.all import *
>>> G = SO(Integer(3),Integer(3))
>>> values  = [Integer(1), -Integer(1), -Integer(1), Integer(1), Integer(2)]
>>> chi = ClassFunction(G, values); chi
Character of Special Orthogonal Group of degree 3 over Finite Field of size 3
>>> loads(dumps(chi)) == chi
True
```

adams_operation(*k*)

Return the `k`-th Adams operation on `self`.

Let G be a finite group. The k-th Adams operation Ψ^k is given by

$$\Psi^k(\chi)(g) = \chi(g^k).$$

The Adams operations turn the representation ring of G into a λ-ring.

EXAMPLES:

```
sage: G = GL(2,3)
sage: chars = G.irreducible_characters()
sage: [chi.adams_operation(2).values() for chi in chars]
[[1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1],
 [2, -1, 2, -1, 2, 2, 2, 2],
 [2, -1, 2, -1, -2, 0, 0, 2],
 [2, -1, 2, -1, -2, 0, 0, 2],
 [3, 0, 3, 0, 3, -1, -1, 3],
```

(continues on next page)

(continued from previous page)

```
 [3, 0, 3, 0, 3, -1, -1, 3],
 [4, 1, 4, 1, -4, 0, 0, 4]]
sage: chars[5].adams_operation(3).decompose()
((1, Character of General Linear Group of degree 2 over Finite Field of size␣
↪3),
 (1, Character of General Linear Group of degree 2 over Finite Field of size␣
↪3),
 (-1, Character of General Linear Group of degree 2 over Finite Field of size␣
↪3),
 (1, Character of General Linear Group of degree 2 over Finite Field of size␣
↪3))
```

```
>>> from sage.all import *
>>> G = GL(Integer(2),Integer(3))
>>> chars = G.irreducible_characters()
>>> [chi.adams_operation(Integer(2)).values() for chi in chars]
[[1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1],
 [2, -1, 2, -1, 2, 2, 2, 2],
 [2, -1, 2, -1, -2, 0, 0, 2],
 [2, -1, 2, -1, -2, 0, 0, 2],
 [3, 0, 3, 0, 3, -1, -1, 3],
 [3, 0, 3, 0, 3, -1, -1, 3],
 [4, 1, 4, 1, -4, 0, 0, 4]]
>>> chars[Integer(5)].adams_operation(Integer(3)).decompose()
((1, Character of General Linear Group of degree 2 over Finite Field of size␣
↪3),
 (1, Character of General Linear Group of degree 2 over Finite Field of size␣
↪3),
 (-1, Character of General Linear Group of degree 2 over Finite Field of size␣
↪3),
 (1, Character of General Linear Group of degree 2 over Finite Field of size␣
↪3))
```

REFERENCES:

- Wikipedia article Adams_operation

central_character()

Return the central character of `self`.

EXAMPLES:

```
sage: t = SymmetricGroup(4).trivial_character()
sage: t.central_character().values()
[1, 6, 3, 8, 6]
```

```
>>> from sage.all import *
>>> t = SymmetricGroup(Integer(4)).trivial_character()
>>> t.central_character().values()
[1, 6, 3, 8, 6]
```

decompose()

Return a list of the characters appearing the decomposition of `self`.

EXAMPLES:

```
sage: S5 = SymmetricGroup(5)
sage: chi = ClassFunction(S5, [22, -8, 2, 1, 1, 2, -3])
sage: chi.decompose()
((3, Character of Symmetric group of order 5! as a permutation group),
 (2, Character of Symmetric group of order 5! as a permutation group))
```

```
>>> from sage.all import *
>>> S5 = SymmetricGroup(Integer(5))
>>> chi = ClassFunction(S5, [Integer(22), -Integer(8), Integer(2), Integer(1),
↪ Integer(1), Integer(2), -Integer(3)])
>>> chi.decompose()
((3, Character of Symmetric group of order 5! as a permutation group),
 (2, Character of Symmetric group of order 5! as a permutation group))
```

degree()

Return the degree of the character `self`.

EXAMPLES:

```
sage: S5 = SymmetricGroup(5)
sage: irr = S5.irreducible_characters()
sage: [x.degree() for x in irr]
[1, 4, 5, 6, 5, 4, 1]
```

```
>>> from sage.all import *
>>> S5 = SymmetricGroup(Integer(5))
>>> irr = S5.irreducible_characters()
>>> [x.degree() for x in irr]
[1, 4, 5, 6, 5, 4, 1]
```

determinant_character()

Return the determinant character of `self`.

EXAMPLES:

```
sage: t = ClassFunction(SymmetricGroup(4), [1, -1, 1, 1, -1])
sage: t.determinant_character().values()
[1, -1, 1, 1, -1]
```

```
>>> from sage.all import *
>>> t = ClassFunction(SymmetricGroup(Integer(4)), [Integer(1), -Integer(1),
↪Integer(1), Integer(1), -Integer(1)])
>>> t.determinant_character().values()
[1, -1, 1, 1, -1]
```

domain()

Return the domain of `self`.

OUTPUT: the underlying group of the class function

EXAMPLES:

```
sage: ClassFunction(SymmetricGroup(4), [1,-1,1,1,-1]).domain()
Symmetric group of order 4! as a permutation group
```

```
>>> from sage.all import *
>>> ClassFunction(SymmetricGroup(Integer(4)), [Integer(1),-Integer(1),
↪Integer(1),Integer(1),-Integer(1)]).domain()
Symmetric group of order 4! as a permutation group
```

exterior_power(*n*)

Return the antisymmetrized product of `self` with itself `n` times.

INPUT:

- `n` – positive integer

OUTPUT: the `n`-th antisymmetrized power of `self` as a *ClassFunction*

EXAMPLES:

```
sage: chi = ClassFunction(SymmetricGroup(4), [3, 1, -1, 0, -1])
sage: p = chi.exterior_power(3)    # the highest antisymmetric power for a 3-d␣
↪character
sage: p
Character of Symmetric group of order 4! as a permutation group
sage: p.values()
[1, -1, 1, 1, -1]
sage: p == chi.determinant_character()
True
```

```
>>> from sage.all import *
>>> chi = ClassFunction(SymmetricGroup(Integer(4)), [Integer(3), Integer(1), -
↪Integer(1), Integer(0), -Integer(1)])
>>> p = chi.exterior_power(Integer(3))    # the highest antisymmetric power␣
↪for a 3-d character
>>> p
Character of Symmetric group of order 4! as a permutation group
>>> p.values()
[1, -1, 1, 1, -1]
>>> p == chi.determinant_character()
True
```

gap()

Return the underlying LibGAP element.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: values  = [1, -1, 1, -1]
sage: chi = ClassFunction(G, values);  chi
Character of Cyclic group of order 4 as a permutation group
sage: type(chi)
<class 'sage.groups.class_function.ClassFunction_libgap'>
sage: libgap(chi)
ClassFunction( CharacterTable( Group([ (1,2,3,4) ]) ), [ 1, -1, 1, -1 ] )
```

(continues on next page)

(continued from previous page)

```
sage: type(_)
<class 'sage.libs.gap.element.GapElement_List'>
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(4))
>>> values  = [Integer(1), -Integer(1), Integer(1), -Integer(1)]
>>> chi = ClassFunction(G, values);  chi
Character of Cyclic group of order 4 as a permutation group
>>> type(chi)
<class 'sage.groups.class_function.ClassFunction_libgap'>
>>> libgap(chi)
ClassFunction( CharacterTable( Group([ (1,2,3,4) ]) ), [ 1, -1, 1, -1 ] )
>>> type(_)
<class 'sage.libs.gap.element.GapElement_List'>
```

induct(G)

Return the induced character.

INPUT:

- `G` – a supergroup of the underlying group of `self`

OUTPUT:

A *ClassFunction* of `G` defined by induction. Induction is the adjoint functor to restriction, see *restrict()*.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: H = G.subgroup([(1,2,3), (1,2), (4,5)])
sage: xi = H.trivial_character(); xi
Character of Subgroup generated by [(1,2,3), (1,2), (4,5)] of
 (Symmetric group of order 5! as a permutation group)
sage: xi.induct(G)
Character of Symmetric group of order 5! as a permutation group
sage: xi.induct(G).values()
[10, 4, 2, 1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> H = G.subgroup([(Integer(1),Integer(2),Integer(3)), (Integer(1),
→Integer(2)), (Integer(4),Integer(5))])
>>> xi = H.trivial_character(); xi
Character of Subgroup generated by [(1,2,3), (1,2), (4,5)] of
 (Symmetric group of order 5! as a permutation group)
>>> xi.induct(G)
Character of Symmetric group of order 5! as a permutation group
>>> xi.induct(G).values()
[10, 4, 2, 1, 1, 0, 0]
```

irreducible_constituents()

Return a list of the characters that appear in the decomposition of `self`.

EXAMPLES:

```
sage: S5 = SymmetricGroup(5)
sage: chi = ClassFunction(S5, [22, -8, 2, 1, 1, 2, -3])
sage: irr = chi.irreducible_constituents(); irr
(Character of Symmetric group of order 5! as a permutation group,
 Character of Symmetric group of order 5! as a permutation group)
sage: list(map(list, irr))
[[4, -2, 0, 1, 1, 0, -1], [5, -1, 1, -1, -1, 1, 0]]

sage: G = GL(2,3)
sage: chi = ClassFunction(G, [-1, -1, -1, -1, -1, -1, -1, -1])
sage: chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
sage: chi = ClassFunction(G, [1, 1, 1, 1, 1, 1, 1, 1])
sage: chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
sage: chi = ClassFunction(G, [2, 2, 2, 2, 2, 2, 2, 2])
sage: chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
sage: chi = ClassFunction(G, [-1, -1, -1, -1, 3, -1, -1, 1])
sage: ic = chi.irreducible_constituents(); ic
(Character of General Linear Group of degree 2 over Finite Field of size 3,
 Character of General Linear Group of degree 2 over Finite Field of size 3)
sage: list(map(list, ic))
[[2, -1, 2, -1, 2, 0, 0, 0], [3, 0, 3, 0, -1, 1, 1, -1]]
```

```
>>> from sage.all import *
>>> S5 = SymmetricGroup(Integer(5))
>>> chi = ClassFunction(S5, [Integer(22), -Integer(8), Integer(2), Integer(1),
↪ Integer(1), Integer(2), -Integer(3)])
>>> irr = chi.irreducible_constituents(); irr
(Character of Symmetric group of order 5! as a permutation group,
 Character of Symmetric group of order 5! as a permutation group)
>>> list(map(list, irr))
[[4, -2, 0, 1, 1, 0, -1], [5, -1, 1, -1, -1, 1, 0]]

>>> G = GL(Integer(2),Integer(3))
>>> chi = ClassFunction(G, [-Integer(1), -Integer(1), -Integer(1), -
↪Integer(1), -Integer(1), -Integer(1), -Integer(1), -Integer(1)])
>>> chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
>>> chi = ClassFunction(G, [Integer(1), Integer(1), Integer(1), Integer(1),␣
↪Integer(1), Integer(1), Integer(1), Integer(1)])
>>> chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
>>> chi = ClassFunction(G, [Integer(2), Integer(2), Integer(2), Integer(2),␣
↪Integer(2), Integer(2), Integer(2), Integer(2)])
>>> chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
>>> chi = ClassFunction(G, [-Integer(1), -Integer(1), -Integer(1), -
↪Integer(1), Integer(3), -Integer(1), -Integer(1), Integer(1)])
>>> ic = chi.irreducible_constituents(); ic
(Character of General Linear Group of degree 2 over Finite Field of size 3,
```

(continues on next page)

(continued from previous page)

```
 Character of General Linear Group of degree 2 over Finite Field of size 3)
>>> list(map(list, ic))
[[2, -1, 2, -1, 2, 0, 0, 0], [3, 0, 3, 0, -1, 1, 1, -1]]
```

is_irreducible()

Return `True` if `self` cannot be written as the sum of two nonzero characters of `self`.

EXAMPLES:

```
sage: S4 = SymmetricGroup(4)
sage: irr = S4.irreducible_characters()
sage: [x.is_irreducible() for x in irr]
[True, True, True, True, True]
```

```
>>> from sage.all import *
>>> S4 = SymmetricGroup(Integer(4))
>>> irr = S4.irreducible_characters()
>>> [x.is_irreducible() for x in irr]
[True, True, True, True, True]
```

norm()

Return the norm of `self`.

EXAMPLES:

```
sage: A5 = AlternatingGroup(5)
sage: [x.norm() for x in A5.irreducible_characters()]
[1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> A5 = AlternatingGroup(Integer(5))
>>> [x.norm() for x in A5.irreducible_characters()]
[1, 1, 1, 1, 1]
```

restrict(*H*)

Return the restricted character.

INPUT:

- `H` – a subgroup of the underlying group of `self`

OUTPUT: a *ClassFunction* of `H` defined by restriction

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: chi = ClassFunction(G, [3, -3, -1, 0, 0, -1, 3]); chi
Character of Symmetric group of order 5! as a permutation group
sage: H = G.subgroup([(1,2,3), (1,2), (4,5)])
sage: chi.restrict(H)
Character of Subgroup generated by [(1,2,3), (1,2), (4,5)] of
 (Symmetric group of order 5! as a permutation group)
sage: chi.restrict(H).values()
[3, -3, -3, -1, 0, 0]
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> chi = ClassFunction(G, [Integer(3), -Integer(3), -Integer(1), Integer(0),␣
→Integer(0), -Integer(1), Integer(3)]); chi
Character of Symmetric group of order 5! as a permutation group
>>> H = G.subgroup([(Integer(1),Integer(2),Integer(3)), (Integer(1),
→Integer(2)), (Integer(4),Integer(5))])
>>> chi.restrict(H)
Character of Subgroup generated by [(1,2,3), (1,2), (4,5)] of
 (Symmetric group of order 5! as a permutation group)
>>> chi.restrict(H).values()
[3, -3, -3, -1, 0, 0]
```

scalar_product(*other*)

Return the scalar product of `self` with `other`.

EXAMPLES:

```
sage: S4 = SymmetricGroup(4)
sage: irr = S4.irreducible_characters()
sage: [[x.scalar_product(y) for x in irr] for y in irr]
[[1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 1]]
```

```
>>> from sage.all import *
>>> S4 = SymmetricGroup(Integer(4))
>>> irr = S4.irreducible_characters()
>>> [[x.scalar_product(y) for x in irr] for y in irr]
[[1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 1]]
```

symmetric_power(*n*)

Return the symmetrized product of `self` with itself `n` times.

INPUT:

- `n` – positive integer

OUTPUT: the `n`-th symmetrized power of `self` as a *ClassFunction*

EXAMPLES:

```
sage: chi = ClassFunction(SymmetricGroup(4), [3, 1, -1, 0, -1])
sage: p = chi.symmetric_power(3)
sage: p
Character of Symmetric group of order 4! as a permutation group
sage: p.values()
[10, 2, -2, 1, 0]
```

```
>>> from sage.all import *
>>> chi = ClassFunction(SymmetricGroup(Integer(4)), [Integer(3), Integer(1), -
↪Integer(1), Integer(0), -Integer(1)])
>>> p = chi.symmetric_power(Integer(3))
>>> p
Character of Symmetric group of order 4! as a permutation group
>>> p.values()
[10, 2, -2, 1, 0]
```

tensor_product(*other*)

Return the tensor product of `self` and `other`.

EXAMPLES:

```
sage: S3 = SymmetricGroup(3)
sage: chi1, chi2, chi3 = S3.irreducible_characters()
sage: chi1.tensor_product(chi3).values()
[1, -1, 1]
```

```
>>> from sage.all import *
>>> S3 = SymmetricGroup(Integer(3))
>>> chi1, chi2, chi3 = S3.irreducible_characters()
>>> chi1.tensor_product(chi3).values()
[1, -1, 1]
```

values()

Return the list of values of `self` on the conjugacy classes.

EXAMPLES:

```
sage: G = GL(2,3)
sage: [x.values() for x in G.irreducible_characters()]  # random
[[1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, -1, -1, -1],
 [2, -1, 2, -1, 2, 0, 0, 0],
 [2, 1, -2, -1, 0, -zeta8^3 - zeta8, zeta8^3 + zeta8, 0],
 [2, 1, -2, -1, 0, zeta8^3 + zeta8, -zeta8^3 - zeta8, 0],
 [3, 0, 3, 0, -1, -1, -1, 1],
 [3, 0, 3, 0, -1, 1, 1, -1],
 [4, -1, -4, 1, 0, 0, 0, 0]]
```

```
>>> from sage.all import *
>>> G = GL(Integer(2),Integer(3))
>>> [x.values() for x in G.irreducible_characters()]  # random
[[1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, -1, -1, -1],
 [2, -1, 2, -1, 2, 0, 0, 0],
 [2, 1, -2, -1, 0, -zeta8^3 - zeta8, zeta8^3 + zeta8, 0],
 [2, 1, -2, -1, 0, zeta8^3 + zeta8, -zeta8^3 - zeta8, 0],
 [3, 0, 3, 0, -1, -1, -1, 1],
 [3, 0, 3, 0, -1, 1, 1, -1],
 [4, -1, -4, 1, 0, 0, 0, 0]]
```

CHAPTER

TWENTYFIVE

CONJUGACY CLASSES OF GROUPS

This module implements a wrapper of GAP's `ConjugacyClass` function.

There are two main classes, `ConjugacyClass` and `ConjugacyClassGAP`. All generic methods should go into `ConjugacyClass`, whereas `ConjugacyClassGAP` should only contain wrappers for GAP functions. `ConjugacyClass` contains some fallback methods in case some group cannot be defined as a GAP object.

Todo

- Implement a non-naive fallback method for computing all the elements of the conjugacy class when the group is not defined in GAP, as the one in Butler's paper.
- Define a sage method for gap matrices so that groups of matrices can use the quicker GAP algorithm rather than the naive one.

EXAMPLES:

Conjugacy classes for groups of permutations:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: G.conjugacy_class(g)                                                    #␣
↪needs sage.combinat
Conjugacy class of cycle type [4] in Symmetric group of order 4! as a permutation␣
↪group
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
>>> g = G((Integer(1),Integer(2),Integer(3),Integer(4)))
>>> G.conjugacy_class(g)                                                    #␣
↪needs sage.combinat
Conjugacy class of cycle type [4] in Symmetric group of order 4! as a permutation␣
↪group
```

Conjugacy classes for groups of matrices:

```
sage: F = GF(5)
sage: gens = [matrix(F,2,[1,2, -1, 1]), matrix(F,2, [1,1, 0,1])]
sage: H = MatrixGroup(gens)
sage: h = H(matrix(F,2,[1,2, -1, 1]))
sage: H.conjugacy_class(h)
Conjugacy class of [1 2]
```

(continues on next page)

(continued from previous page)

```
[4 1] in Matrix group over Finite Field of size 5 with 2 generators (
[1 2]   [1 1]
[4 1], [0 1]
)
```

```
>>> from sage.all import *
>>> F = GF(Integer(5))
>>> gens = [matrix(F,Integer(2),[Integer(1),Integer(2), -Integer(1), Integer(1)]),␣
↪matrix(F,Integer(2), [Integer(1),Integer(1), Integer(0),Integer(1)])]
>>> H = MatrixGroup(gens)
>>> h = H(matrix(F,Integer(2),[Integer(1),Integer(2), -Integer(1), Integer(1)]))
>>> H.conjugacy_class(h)
Conjugacy class of [1 2]
[4 1] in Matrix group over Finite Field of size 5 with 2 generators (
[1 2]   [1 1]
[4 1], [0 1]
)
```

class sage.groups.conjugacy_classes.**ConjugacyClass**(*group*, *element*)

Bases: Parent

Generic conjugacy classes for elements in a group.

This is the default fall-back implementation to be used whenever GAP cannot handle the group.

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: ConjugacyClass(G,g)
Conjugacy class of (1,2,3,4) in Symmetric group of order 4! as a
permutation group
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
>>> g = G((Integer(1),Integer(2),Integer(3),Integer(4)))
>>> ConjugacyClass(G,g)
Conjugacy class of (1,2,3,4) in Symmetric group of order 4! as a
permutation group
```

an_element()

Return a representative of `self`.

EXAMPLES:

```
sage: G = SymmetricGroup(3)
sage: g = G((1,2,3))
sage: C = ConjugacyClass(G,g)
sage: C.representative()
(1,2,3)
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> g = G((Integer(1),Integer(2),Integer(3)))
>>> C = ConjugacyClass(G,g)
>>> C.representative()
(1,2,3)
```

is_rational()

Check if `self` is rational (closed for powers).

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: c = ConjugacyClass(G,g)
sage: c.is_rational()
False
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
>>> g = G((Integer(1),Integer(2),Integer(3),Integer(4)))
>>> c = ConjugacyClass(G,g)
>>> c.is_rational()
False
```

is_real()

Check if `self` is real (closed for inverses).

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: c = ConjugacyClass(G,g)
sage: c.is_real()
True
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
>>> g = G((Integer(1),Integer(2),Integer(3),Integer(4)))
>>> c = ConjugacyClass(G,g)
>>> c.is_real()
True
```

list()

Return a list with all the elements of `self`.

EXAMPLES:

Groups of permutations:

```
sage: G = SymmetricGroup(3)
sage: g = G((1,2,3))
sage: c = ConjugacyClass(G,g)
sage: L = c.list()
sage: Set(L) == Set([G((1,3,2)), G((1,2,3))])
True
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> g = G((Integer(1),Integer(2),Integer(3)))
>>> c = ConjugacyClass(G,g)
>>> L = c.list()
>>> Set(L) == Set([G((Integer(1),Integer(3),Integer(2))), G((Integer(1),
↪Integer(2),Integer(3)))])
True
```

representative()

Return a representative of `self`.

EXAMPLES:

```
sage: G = SymmetricGroup(3)
sage: g = G((1,2,3))
sage: C = ConjugacyClass(G,g)
sage: C.representative()
(1,2,3)
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> g = G((Integer(1),Integer(2),Integer(3)))
>>> C = ConjugacyClass(G,g)
>>> C.representative()
(1,2,3)
```

set()

Return the set of elements of the conjugacy class.

EXAMPLES:

Groups of permutations:

```
sage: G = SymmetricGroup(3)
sage: g = G((1,2))
sage: C = ConjugacyClass(G,g)
sage: S = [(2,3), (1,2), (1,3)]
sage: C.set() == Set(G(x) for x in S)
True
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> g = G((Integer(1),Integer(2)))
>>> C = ConjugacyClass(G,g)
>>> S = [(Integer(2),Integer(3)), (Integer(1),Integer(2)), (Integer(1),
↪Integer(3))]
>>> C.set() == Set(G(x) for x in S)
True
```

Groups of matrices over finite fields:

```
sage: F = GF(5)
sage: gens = [matrix(F,2,[1,2, -1, 1]), matrix(F,2, [1,1, 0,1])]
```

(continues on next page)

(continued from previous page)

```
sage: H = MatrixGroup(gens)
sage: h = H(matrix(F,2,[1,2, -1, 1]))
sage: C = ConjugacyClass(H,h)
sage: S = [[[3, 2], [2, 4]], [[0, 1], [2, 2]], [[3, 4], [1, 4]],\
....: [[0, 3], [4, 2]], [[1, 2], [4, 1]], [[2, 1], [2, 0]],\
....: [[4, 1], [4, 3]], [[4, 4], [1, 3]], [[2, 4], [3, 0]],\
....: [[1, 4], [2, 1]], [[3, 3], [3, 4]], [[2, 3], [4, 0]],\
....: [[0, 2], [1, 2]], [[1, 3], [1, 1]], [[4, 3], [3, 3]],\
....: [[4, 2], [2, 3]], [[0, 4], [3, 2]], [[1, 1], [3, 1]],\
....: [[2, 2], [1, 0]], [[3, 1], [4, 4]]]
sage: C.set() == Set(H(x) for x in S)
True
```

```
>>> from sage.all import *
>>> F = GF(Integer(5))
>>> gens = [matrix(F,Integer(2),[Integer(1),Integer(2), -Integer(1),␣
↪Integer(1)]), matrix(F,Integer(2), [Integer(1),Integer(1), Integer(0),
↪Integer(1)])]
>>> H = MatrixGroup(gens)
>>> h = H(matrix(F,Integer(2),[Integer(1),Integer(2), -Integer(1),␣
↪Integer(1)]))
>>> C = ConjugacyClass(H,h)
>>> S = [[[Integer(3), Integer(2)], [Integer(2), Integer(4)]], [[Integer(0),␣
↪Integer(1)], [Integer(2), Integer(2)]], [[Integer(3), Integer(4)],␣
↪[Integer(1), Integer(4)]],[[Integer(0), Integer(3)], [Integer(4),␣
↪Integer(2)]], [[Integer(1), Integer(2)], [Integer(4), Integer(1)]],␣
↪[[Integer(2), Integer(1)], [Integer(2), Integer(0)]],[[Integer(4),␣
↪Integer(1)], [Integer(4), Integer(3)]], [[Integer(4), Integer(4)],␣
↪[Integer(1), Integer(3)]], [[Integer(2), Integer(4)], [Integer(3),␣
↪Integer(0)]],[[Integer(1), Integer(4)], [Integer(2), Integer(1)]],␣
↪[[Integer(3), Integer(3)], [Integer(3), Integer(4)]], [[Integer(2),␣
↪Integer(3)], [Integer(4), Integer(0)]],[[Integer(0), Integer(2)],␣
↪[Integer(1), Integer(2)]], [[Integer(1), Integer(3)], [Integer(1),␣
↪Integer(1)]], [[Integer(4), Integer(3)], [Integer(3), Integer(3)]],
↪[[Integer(4), Integer(2)], [Integer(2), Integer(3)]], [[Integer(0),␣
↪Integer(4)], [Integer(3), Integer(2)]], [[Integer(1), Integer(1)],␣
↪[Integer(3), Integer(1)]],[[Integer(2), Integer(2)], [Integer(1),␣
↪Integer(0)]], [[Integer(3), Integer(1)], [Integer(4), Integer(4)]]]
>>> C.set() == Set(H(x) for x in S)
True
```

It is not implemented for infinite groups:

```
sage: a = matrix(ZZ,2,[1,1,0,1])
sage: b = matrix(ZZ,2,[1,0,1,1])
sage: G = MatrixGroup([a,b])        # takes 1s
sage: g = G(a)
sage: C = ConjugacyClass(G, g)
sage: C.set()
Traceback (most recent call last):
...
NotImplementedError: Listing the elements of conjugacy classes is not␣
```

(continues on next page)

(continued from previous page)

```
↪implemented for infinite groups! Use the iter function instead.
```

```
>>> from sage.all import *
>>> a = matrix(ZZ,Integer(2),[Integer(1),Integer(1),Integer(0),Integer(1)])
>>> b = matrix(ZZ,Integer(2),[Integer(1),Integer(0),Integer(1),Integer(1)])
>>> G = MatrixGroup([a,b])        # takes 1s
>>> g = G(a)
>>> C = ConjugacyClass(G, g)
>>> C.set()
Traceback (most recent call last):
...
NotImplementedError: Listing the elements of conjugacy classes is not␣
↪implemented for infinite groups! Use the iter function instead.
```

class sage.groups.conjugacy_classes.**ConjugacyClassGAP**(*group*, *element*)

Bases: *ConjugacyClass*

Class for a conjugacy class for groups defined over GAP.

Intended for wrapping GAP methods on conjugacy classes.

INPUT:

- group – the group in which the conjugacy class is taken
- element – the element generating the conjugacy class

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: ConjugacyClassGAP(G,g)
Conjugacy class of (1,2,3,4) in Symmetric group of order 4! as a
permutation group
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
>>> g = G((Integer(1),Integer(2),Integer(3),Integer(4)))
>>> ConjugacyClassGAP(G,g)
Conjugacy class of (1,2,3,4) in Symmetric group of order 4! as a
permutation group
```

cardinality()

Return the size of this conjugacy class.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: W = WeylGroup(['C',6])
sage: cc = W.conjugacy_class(W.an_element())
sage: cc.cardinality()
3840
sage: type(cc.cardinality())
<class 'sage.rings.integer.Integer'>
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> W = WeylGroup(['C',Integer(6)])
>>> cc = W.conjugacy_class(W.an_element())
>>> cc.cardinality()
3840
>>> type(cc.cardinality())
<class 'sage.rings.integer.Integer'>
```

set ()

Return a Sage `Set` with all the elements of the conjugacy class.

By default attempts to use GAP construction of the conjugacy class. If GAP method is not implemented for the given group, and the group is finite, falls back to a naive algorithm.

Warning: The naive algorithm can be really slow and memory intensive.

EXAMPLES:

Groups of permutations:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: C = ConjugacyClassGAP(G,g)
sage: S = [(1,3,2,4), (1,4,3,2), (1,3,4,2), (1,2,3,4), (1,4,2,3), (1,2,4,3)]
sage: C.set() == Set(G(x) for x in S)
True
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
>>> g = G((Integer(1),Integer(2),Integer(3),Integer(4)))
>>> C = ConjugacyClassGAP(G,g)
>>> S = [(Integer(1),Integer(3),Integer(2),Integer(4)), (Integer(1),
↪Integer(4),Integer(3),Integer(2)), (Integer(1),Integer(3),Integer(4),
↪Integer(2)), (Integer(1),Integer(2),Integer(3),Integer(4)), (Integer(1),
↪Integer(4),Integer(2),Integer(3)), (Integer(1),Integer(2),Integer(4),
↪Integer(3))]
>>> C.set() == Set(G(x) for x in S)
True
```

CHAPTER

TWENTYSIX

ABELIAN GROUPS

26.1 Multiplicative Abelian Groups

This module lets you compute with finitely generated Abelian groups of the form

$$G = \mathbf{Z}^r \oplus \mathbf{Z}_{k_1} \oplus \cdots \oplus \mathbf{Z}_{k_t}$$

It is customary to denote the infinite cyclic group $\mathbf{Z}$ as having order 0, so the data defining the Abelian group can be written as an integer vector

$$\vec{k} = (0, \ldots, 0, k_1, \ldots, k_t)$$

where there are r zeroes and t nonzero values. To construct this Abelian group in Sage, you can either specify all entries of $\vec{k}$ or only the nonzero entries together with the total number of generators:

```
sage: AbelianGroup([0,0,0,2,3])
Multiplicative Abelian group isomorphic to Z x Z x Z x C2 x C3
sage: AbelianGroup(5, [2,3])
Multiplicative Abelian group isomorphic to Z x Z x Z x C2 x C3
```

```
>>> from sage.all import *
>>> AbelianGroup([Integer(0),Integer(0),Integer(0),Integer(2),Integer(3)])
Multiplicative Abelian group isomorphic to Z x Z x Z x C2 x C3
>>> AbelianGroup(Integer(5), [Integer(2),Integer(3)])
Multiplicative Abelian group isomorphic to Z x Z x Z x C2 x C3
```

It is also legal to specify 1 as the order. The corresponding generator will be the neutral element, but it will still take up an index in the labelling of the generators:

```
sage: G = AbelianGroup([2,1,3], names='g')
sage: G.gens()
(g0, 1, g2)
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(2),Integer(1),Integer(3)], names='g')
>>> G.gens()
(g0, 1, g2)
```

Note that this presentation is not unique, for example $\mathbf{Z}_6 \cong \mathbf{Z}_2 \times \mathbf{Z}_3$. The orders of the generators $\vec{k} = (0, \ldots, 0, k_1, \ldots, k_t)$ has previously been called invariants in Sage, even though they are not necessarily the (unique) invariant factors of the group. You should now use `gens_orders()` instead:

```
sage: J = AbelianGroup([2,0,3,2,4]);  J
Multiplicative Abelian group isomorphic to C2 x Z x C3 x C2 x C4
sage: J.gens_orders()            # use this instead
(2, 0, 3, 2, 4)
sage: J.invariants()             # deprecated
(2, 0, 3, 2, 4)
sage: J.elementary_divisors()    # these are the "invariant factors"
(2, 2, 12, 0)
sage: for i in range(J.ngens()):
....:     print((i, J.gen(i), J.gen(i).order()))     # or use this form
(0, f0, 2)
(1, f1, +Infinity)
(2, f2, 3)
(3, f3, 2)
(4, f4, 4)
```

```
>>> from sage.all import *
>>> J = AbelianGroup([Integer(2),Integer(0),Integer(3),Integer(2),Integer(4)]);  J
Multiplicative Abelian group isomorphic to C2 x Z x C3 x C2 x C4
>>> J.gens_orders()            # use this instead
(2, 0, 3, 2, 4)
>>> J.invariants()             # deprecated
(2, 0, 3, 2, 4)
>>> J.elementary_divisors()    # these are the "invariant factors"
(2, 2, 12, 0)
>>> for i in range(J.ngens()):
...     print((i, J.gen(i), J.gen(i).order()))     # or use this form
(0, f0, 2)
(1, f1, +Infinity)
(2, f2, 3)
(3, f3, 2)
(4, f4, 4)
```

Background on invariant factors and the Smith normal form (according to section 4.1 of [Cohen1]): An abelian group is a group A for which there exists an exact sequence $\mathbf{Z}^k \to \mathbf{Z}^\ell \to A \to 1$, for some positive integers k, ℓ with $k \leq \ell$. For example, a finite abelian group has a decomposition

$$A = \langle a_1 \rangle \times \cdots \times \langle a_\ell \rangle,$$

where $\mathrm{ord}(a_i) = p_i^{c_i}$, for some primes p_i and some positive integers c_i, $i = 1, ..., \ell$. GAP calls the list (ordered by size) of the $p_i^{c_i}$ the *abelian invariants*. In Sage they will be called *invariants*. In this situation, $k = \ell$ and $\phi : \mathbf{Z}^\ell \to A$ is the map $\phi(x_1, ..., x_\ell) = a_1^{x_1} ... a_\ell^{x_\ell}$, for $(x_1, ..., x_\ell) \in \mathbf{Z}^\ell$. The matrix of relations $M : \mathbf{Z}^k \to \mathbf{Z}^\ell$ is the matrix whose rows generate the kernel of ϕ as a $\mathbf{Z}$-module. In other words, $M = (M_{ij})$ is a $\ell \times \ell$ diagonal matrix with $M_{ii} = p_i^{c_i}$. Consider now the subgroup $B \subset A$ generated by $b_1 = a_1^{f_{1,1}} ... a_\ell^{f_{\ell,1}}, \ldots, b_m = a_1^{f_{1,m}} ... a_\ell^{f_{\ell,m}}$. The kernel of the map $\phi_B : \mathbf{Z}^m \to B$ defined by $\phi_B(y_1, ..., y_m) = b_1^{y_1} ... b_m^{y_m}$, for $(y_1, ..., y_m) \in \mathbf{Z}^m$, is the kernel of the matrix

$$F = \begin{pmatrix} f_{11} & f_{12} & \dots & f_{1m} \\ f_{21} & f_{22} & \dots & f_{2m} \\ \vdots & & \ddots & \vdots \\ f_{\ell,1} & f_{\ell,2} & \dots & f_{\ell,m} \end{pmatrix},$$

regarded as a map $\mathbf{Z}^m \to (\mathbf{Z}/p_1^{c_1}\mathbf{Z}) \times ... \times (\mathbf{Z}/p_\ell^{c_\ell}\mathbf{Z})$. In particular, $B \cong \mathbf{Z}^m / \ker(F)$. If $B = A$ then the Smith normal form (SNF) of a generator matrix of $\ker(F)$ and the SNF of M are the same. The diagonal entries s_i of the SNF

$S = \text{diag}[s_1, s_2, s_3, ...s_r, 0, 0, ...0]$, are called *determinantal divisors* of F. where r is the rank. The *invariant factors* of A are:

$$s_1, s_2/s_1, s_3/s_2, ...s_r/s_{r-1}.$$

Sage supports multiplicative abelian groups on any prescribed finite number $n \geq 0$ of generators. Use the `AbelianGroup()` function to create an abelian group, and the `gen()` and `gens()` methods to obtain the corresponding generators. You can print the generators as arbitrary strings using the optional `names` argument to the `AbelianGroup()` function.

EXAMPLE 1:

We create an abelian group in zero or more variables; the syntax `T(1)` creates the identity element even in the rank zero case:

```
sage: T = AbelianGroup(0, [])
sage: T
Trivial Abelian group
sage: T.gens()
()
sage: T(1)
1
```

```
>>> from sage.all import *
>>> T = AbelianGroup(Integer(0), [])
>>> T
Trivial Abelian group
>>> T.gens()
()
>>> T(Integer(1))
1
```

EXAMPLE 2:

An Abelian group uses a multiplicative representation of elements, but the underlying representation is lists of integer exponents:

```
sage: F = AbelianGroup(5, [3,4,5,5,7], names = list("abcde"))
sage: F
Multiplicative Abelian group isomorphic to C3 x C4 x C5 x C5 x C7
sage: (a,b,c,d,e) = F.gens()
sage: a*b^2*e*d
a*b^2*d*e
sage: x = b^2*e*d*a^7
sage: x
a*b^2*d*e
sage: x.list()
[1, 2, 0, 1, 1]
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(5), [Integer(3),Integer(4),Integer(5),Integer(5),
↪Integer(7)], names = list("abcde"))
>>> F
Multiplicative Abelian group isomorphic to C3 x C4 x C5 x C5 x C7
>>> (a,b,c,d,e) = F.gens()
>>> a*b**Integer(2)*e*d
```

(continues on next page)

(continued from previous page)

```
a*b^2*d*e
>>> x = b**Integer(2)*e*d*a**Integer(7)
>>> x
a*b^2*d*e
>>> x.list()
[1, 2, 0, 1, 1]
```

REFERENCES:

Warning

Many basic properties for infinite abelian groups are not implemented.

AUTHORS:

- William Stein, David Joyner (2008-12): added (user requested) is_cyclic, fixed elementary_divisors.
- David Joyner (2006-03): (based on free abelian monoids by David Kohel)
- David Joyner (2006-05) several significant bug fixes
- David Joyner (2006-08) trivial changes to docs, added random, fixed bug in how invariants are recorded
- David Joyner (2006-10) added dual_group method
- David Joyner (2008-02) fixed serious bug in word_problem
- David Joyner (2008-03) fixed bug in trivial group case
- David Loeffler (2009-05) added subgroups method
- Volker Braun (2012-11) port to new Parent base. Use tuples for immutables. Rename invariants to gens_orders.

`sage.groups.abelian_gps.abelian_group.`**`AbelianGroup`**`(`*n*, *gens_orders=None*, *names='f'*`)`

Create the multiplicative abelian group in n generators with given orders of generators (which need not be prime powers).

INPUT:

- `n` – integer (optional); if not specified, will be derived from `gens_orders`
- **`gens_orders` – list of nonnegative integers in the form**
  $[a_0, a_1, \dots, a_{n-1}]$, typically written in increasing order. This list is padded with zeros if it has length less than n. The orders of the commuting generators, with 0 denoting an infinite cyclic factor.
- `names` – (optional) names of generators

Alternatively, you can also give input in the form `AbelianGroup(gens_orders, names='f')`, where the names keyword argument must be explicitly named.

OUTPUT:

Abelian group with generators and invariant type. The default name for generator `A.i` is `fi`, as in GAP.

EXAMPLES:

```
sage: F = AbelianGroup(5, [5,5,7,8,9], names='abcde')
sage: F(1)
1
sage: (a, b, c, d, e) = F.gens()
```

(continues on next page)

(continued from previous page)

```
sage: mul([ a, b, a, c, b, d, c, d ], F(1))
a^2*b^2*c^2*d^2
sage: d * b**2 * c**3
b^2*c^3*d
sage: F = AbelianGroup(3, [2]*3); F
Multiplicative Abelian group isomorphic to C2 x C2 x C2
sage: H = AbelianGroup([2,3], names='xy'); H
Multiplicative Abelian group isomorphic to C2 x C3
sage: AbelianGroup(5)
Multiplicative Abelian group isomorphic to Z x Z x Z x Z x Z
sage: AbelianGroup(5).order()
+Infinity
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(5), [Integer(5),Integer(5),Integer(7),Integer(8),
↪Integer(9)], names='abcde')
>>> F(Integer(1))
1
>>> (a, b, c, d, e) = F.gens()
>>> mul([ a, b, a, c, b, d, c, d ], F(Integer(1)))
a^2*b^2*c^2*d^2
>>> d * b**Integer(2) * c**Integer(3)
b^2*c^3*d
>>> F = AbelianGroup(Integer(3), [Integer(2)]*Integer(3)); F
Multiplicative Abelian group isomorphic to C2 x C2 x C2
>>> H = AbelianGroup([Integer(2),Integer(3)], names='xy'); H
Multiplicative Abelian group isomorphic to C2 x C3
>>> AbelianGroup(Integer(5))
Multiplicative Abelian group isomorphic to Z x Z x Z x Z x Z
>>> AbelianGroup(Integer(5)).order()
+Infinity
```

Notice that 0's are prepended if necessary:

```
sage: G = AbelianGroup(5, [2,3,4]);  G
Multiplicative Abelian group isomorphic to Z x Z x C2 x C3 x C4
sage: G.gens_orders()
(0, 0, 2, 3, 4)
```

```
>>> from sage.all import *
>>> G = AbelianGroup(Integer(5), [Integer(2),Integer(3),Integer(4)]);  G
Multiplicative Abelian group isomorphic to Z x Z x C2 x C3 x C4
>>> G.gens_orders()
(0, 0, 2, 3, 4)
```

The invariant list must not be longer than the number of generators:

```
sage: AbelianGroup(2, [2,3,4])
Traceback (most recent call last):
...
ValueError: gens_orders (=(2, 3, 4)) must have length n (=2)
```

```
>>> from sage.all import *
>>> AbelianGroup(Integer(2), [Integer(2),Integer(3),Integer(4)])
Traceback (most recent call last):
...
ValueError: gens_orders (=(2, 3, 4)) must have length n (=2)
```

class sage.groups.abelian_gps.abelian_group.**AbelianGroup_class**(*generator_orders*, *names*, *category=None*)

Bases: `UniqueRepresentation`, `AbelianGroup`

The parent for Abelian groups with chosen generator orders.

> **Warning**
>
> You should use `AbelianGroup()` to construct Abelian groups and not instantiate this class directly.

INPUT:

- `generator_orders` – list of integers. The orders of the (commuting) generators. Zero denotes an infinite cyclic generator.
- `names` – names of the group generators (optional)

EXAMPLES:

```
sage: Z2xZ3 = AbelianGroup([2,3])
sage: Z6 = AbelianGroup([6])
sage: Z2xZ3 is Z2xZ3, Z6 is Z6
(True, True)
sage: Z2xZ3 is Z6
False
sage: Z2xZ3 == Z6
False
sage: Z2xZ3.is_isomorphic(Z6)
True

sage: F = AbelianGroup(5,[5,5,7,8,9], names=list("abcde")); F
Multiplicative Abelian group isomorphic to C5 x C5 x C7 x C8 x C9
sage: F = AbelianGroup(5,[2, 4, 12, 24, 120], names=list("abcde")); F
Multiplicative Abelian group isomorphic to C2 x C4 x C12 x C24 x C120
sage: F.elementary_divisors()
(2, 4, 12, 24, 120)

sage: F.category()
Category of finite enumerated commutative groups
```

```
>>> from sage.all import *
>>> Z2xZ3 = AbelianGroup([Integer(2),Integer(3)])
>>> Z6 = AbelianGroup([Integer(6)])
>>> Z2xZ3 is Z2xZ3, Z6 is Z6
(True, True)
>>> Z2xZ3 is Z6
False
```

(continues on next page)

(continued from previous page)

```
>>> Z2xZ3 == Z6
False
>>> Z2xZ3.is_isomorphic(Z6)
True

>>> F = AbelianGroup(Integer(5),[Integer(5),Integer(5),Integer(7),Integer(8),
↪Integer(9)], names=list("abcde")); F
Multiplicative Abelian group isomorphic to C5 x C5 x C7 x C8 x C9
>>> F = AbelianGroup(Integer(5),[Integer(2), Integer(4), Integer(12), Integer(24),
↪ Integer(120)], names=list("abcde")); F
Multiplicative Abelian group isomorphic to C2 x C4 x C12 x C24 x C120
>>> F.elementary_divisors()
(2, 4, 12, 24, 120)

>>> F.category()
Category of finite enumerated commutative groups
```

Element

alias of *AbelianGroupElement*

Subgroup

alias of *AbelianGroup_subgroup*

cardinality()

Return the order of this group.

EXAMPLES:

```
sage: G = AbelianGroup(2, [2,3])
sage: G.order()
6
sage: G = AbelianGroup(3, [2,3,0])
sage: G.order()
+Infinity
```

```
>>> from sage.all import *
>>> G = AbelianGroup(Integer(2), [Integer(2),Integer(3)])
>>> G.order()
6
>>> G = AbelianGroup(Integer(3), [Integer(2),Integer(3),Integer(0)])
>>> G.order()
+Infinity
```

dual_group(*names='X'*, *base_ring=None*)

Return the dual group.

INPUT:

- `names` – string or list of strings. The generator names for the dual group
- `base_ring` – the base ring; if `None` (default), then a suitable cyclotomic field is picked automatically

OUTPUT: the *dual abelian group*

EXAMPLES:

```
sage: G = AbelianGroup([2])
sage: G.dual_group()                                                           #␣
↪needs sage.rings.number_field
Dual of Abelian Group isomorphic to Z/2Z over Cyclotomic Field of order 2 and␣
↪degree 1
sage: G.dual_group().gens()                                                    #␣
↪needs sage.rings.number_field
(X,)
sage: G.dual_group(names='Z').gens()                                           #␣
↪needs sage.rings.number_field
(Z,)

sage: G.dual_group(base_ring=QQ)
Dual of Abelian Group isomorphic to Z/2Z over Rational Field
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(2)])
>>> G.dual_group()                                                           #␣
↪needs sage.rings.number_field
Dual of Abelian Group isomorphic to Z/2Z over Cyclotomic Field of order 2 and␣
↪degree 1
>>> G.dual_group().gens()                                                    #␣
↪needs sage.rings.number_field
(X,)
>>> G.dual_group(names='Z').gens()                                           #␣
↪needs sage.rings.number_field
(Z,)

>>> G.dual_group(base_ring=QQ)
Dual of Abelian Group isomorphic to Z/2Z over Rational Field
```

elementary_divisors()

Return the elementary divisors of the group, using Pari.

> **Note**
>
> Here is another algorithm for computing the elementary divisors $d_1, d_2, d_3, \ldots$, of a finite abelian group (where $d_1|d_2|d_3|\ldots$ are composed of prime powers dividing the invariants of the group in a way described below). Just factor the invariants a_i that define the abelian group. Then the biggest d_i is the product of the maximum prime powers dividing some a_j. In other words, the largest d_i is the product of p^v, where $v = \max(\mathrm{ord}_p(a_j)$ for all j). Now divide out all those p^v's into the list of invariants a_i, and get a new list of "smaller invariants". Repeat the above procedure on these "smaller invariants" to compute d_{i-1}, and so on. (Thanks to Robert Miller for communicating this algorithm.)

OUTPUT: tuple of integers

EXAMPLES:

```
sage: G = AbelianGroup(2, [2,3])
sage: G.elementary_divisors()
(6,)
sage: G = AbelianGroup(1, [6])
```

(continues on next page)

(continued from previous page)

```
sage: G.elementary_divisors()
(6,)
sage: G = AbelianGroup(2, [2,6])
sage: G
Multiplicative Abelian group isomorphic to C2 x C6
sage: G.gens_orders()
(2, 6)
sage: G.elementary_divisors()
(2, 6)
sage: J = AbelianGroup([1,3,5,12])
sage: J.elementary_divisors()
(3, 60)
sage: G = AbelianGroup(2, [0,6])
sage: G.elementary_divisors()
(6, 0)
sage: AbelianGroup([3,4,5]).elementary_divisors()
(60,)
```

```
>>> from sage.all import *
>>> G = AbelianGroup(Integer(2), [Integer(2),Integer(3)])
>>> G.elementary_divisors()
(6,)
>>> G = AbelianGroup(Integer(1), [Integer(6)])
>>> G.elementary_divisors()
(6,)
>>> G = AbelianGroup(Integer(2), [Integer(2),Integer(6)])
>>> G
Multiplicative Abelian group isomorphic to C2 x C6
>>> G.gens_orders()
(2, 6)
>>> G.elementary_divisors()
(2, 6)
>>> J = AbelianGroup([Integer(1),Integer(3),Integer(5),Integer(12)])
>>> J.elementary_divisors()
(3, 60)
>>> G = AbelianGroup(Integer(2), [Integer(0),Integer(6)])
>>> G.elementary_divisors()
(6, 0)
>>> AbelianGroup([Integer(3),Integer(4),Integer(5)]).elementary_divisors()
(60,)
```

exponent()

Return the exponent of this abelian group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,7]); G
Multiplicative Abelian group isomorphic to C2 x C3 x C7
sage: G.exponent()
42
sage: G = AbelianGroup([2,4,6]); G
Multiplicative Abelian group isomorphic to C2 x C4 x C6
```

(continues on next page)

(continued from previous page)

```
sage: G.exponent()
12
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(2),Integer(3),Integer(7)]); G
Multiplicative Abelian group isomorphic to C2 x C3 x C7
>>> G.exponent()
42
>>> G = AbelianGroup([Integer(2),Integer(4),Integer(6)]); G
Multiplicative Abelian group isomorphic to C2 x C4 x C6
>>> G.exponent()
12
```

gen(*i=0*)

The i-th generator of the abelian group.

EXAMPLES:

```
sage: F = AbelianGroup(5,[],names='a')
sage: F.0
a0
sage: F.2
a2
sage: F.gens_orders()
(0, 0, 0, 0, 0)

sage: G = AbelianGroup([2,1,3])
sage: G.gens()
(f0, 1, f2)
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(5),[],names='a')
>>> F.gen(0)
a0
>>> F.gen(2)
a2
>>> F.gens_orders()
(0, 0, 0, 0, 0)

>>> G = AbelianGroup([Integer(2),Integer(1),Integer(3)])
>>> G.gens()
(f0, 1, f2)
```

gens()

Return the generators of the group.

OUTPUT:

A tuple of group elements. The generators according to the chosen `gens_orders()`.

EXAMPLES:

```
sage: F = AbelianGroup(5, [3,2], names='abcde')
sage: F.gens()
(a, b, c, d, e)
sage: [g.order() for g in F.gens()]
[+Infinity, +Infinity, +Infinity, 3, 2]
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(5), [Integer(3),Integer(2)], names='abcde')
>>> F.gens()
(a, b, c, d, e)
>>> [g.order() for g in F.gens()]
[+Infinity, +Infinity, +Infinity, 3, 2]
```

gens_orders()

Return the orders of the cyclic factors that this group has been defined with.

Use *elementary_divisors()* if you are looking for an invariant of the group.

OUTPUT: tuple of integers

EXAMPLES:

```
sage: Z2xZ3 = AbelianGroup([2,3])
sage: Z2xZ3.gens_orders()
(2, 3)
sage: Z2xZ3.elementary_divisors()
(6,)

sage: Z6 = AbelianGroup([6])
sage: Z6.gens_orders()
(6,)
sage: Z6.elementary_divisors()
(6,)

sage: Z2xZ3.is_isomorphic(Z6)
True
sage: Z2xZ3 is Z6
False
```

```
>>> from sage.all import *
>>> Z2xZ3 = AbelianGroup([Integer(2),Integer(3)])
>>> Z2xZ3.gens_orders()
(2, 3)
>>> Z2xZ3.elementary_divisors()
(6,)

>>> Z6 = AbelianGroup([Integer(6)])
>>> Z6.gens_orders()
(6,)
>>> Z6.elementary_divisors()
(6,)

>>> Z2xZ3.is_isomorphic(Z6)
True
```

(continues on next page)

(continued from previous page)

```
>>> Z2xZ3 is Z6
False
```

identity()

Return the identity element of this group.

EXAMPLES:

```
sage: G = AbelianGroup([2,2])
sage: e = G.identity()
sage: e
1
sage: g = G.gen(0)
sage: g*e
f0
sage: e*g
f0
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(2),Integer(2)])
>>> e = G.identity()
>>> e
1
>>> g = G.gen(Integer(0))
>>> g*e
f0
>>> e*g
f0
```

invariants()

Return the orders of the cyclic factors that this group has been defined with.

For historical reasons this has been called invariants in Sage, even though they are not necessarily the invariant factors of the group. Use *gens_orders()* instead:

```
sage: J = AbelianGroup([2,0,3,2,4]);  J
Multiplicative Abelian group isomorphic to C2 x Z x C3 x C2 x C4
sage: J.invariants()    # deprecated
(2, 0, 3, 2, 4)
sage: J.gens_orders()   # use this instead
(2, 0, 3, 2, 4)
sage: for i in range(J.ngens()):
....:     print((i, J.gen(i), J.gen(i).order()))     # or this
(0, f0, 2)
(1, f1, +Infinity)
(2, f2, 3)
(3, f3, 2)
(4, f4, 4)
```

```
>>> from sage.all import *
>>> J = AbelianGroup([Integer(2),Integer(0),Integer(3),Integer(2),
→Integer(4)]);  J
```

(continues on next page)

(continued from previous page)

```
Multiplicative Abelian group isomorphic to C2 x Z x C3 x C2 x C4
>>> J.invariants()    # deprecated
(2, 0, 3, 2, 4)
>>> J.gens_orders()   # use this instead
(2, 0, 3, 2, 4)
>>> for i in range(J.ngens()):
...     print((i, J.gen(i), J.gen(i).order()))     # or this
(0, f0, 2)
(1, f1, +Infinity)
(2, f2, 3)
(3, f3, 2)
(4, f4, 4)
```

Use *elementary_divisors()* if you are looking for an invariant of the group.

OUTPUT: tuple of integers; zero means infinite cyclic factor

EXAMPLES:

```
sage: J = AbelianGroup([2,3])
sage: J.invariants()
(2, 3)
sage: J.elementary_divisors()
(6,)
```

```
>>> from sage.all import *
>>> J = AbelianGroup([Integer(2),Integer(3)])
>>> J.invariants()
(2, 3)
>>> J.elementary_divisors()
(6,)
```

is_commutative()

Return True since this group is commutative.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,9, 0])
sage: G.is_commutative()
True
sage: G.is_abelian()
True
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(2),Integer(3),Integer(9), Integer(0)])
>>> G.is_commutative()
True
>>> G.is_abelian()
True
```

is_cyclic()

Return True if the group is a cyclic group.

EXAMPLES:

```
sage: J = AbelianGroup([2,3])
sage: J.gens_orders()
(2, 3)
sage: J.elementary_divisors()
(6,)
sage: J.is_cyclic()
True
sage: G = AbelianGroup([6])
sage: G.gens_orders()
(6,)
sage: G.is_cyclic()
True
sage: H = AbelianGroup([2,2])
sage: H.gens_orders()
(2, 2)
sage: H.is_cyclic()
False
sage: H = AbelianGroup([2,4])
sage: H.elementary_divisors()
(2, 4)
sage: H.is_cyclic()
False
sage: H.permutation_group().is_cyclic()                                    #␣
↪needs sage.groups
False
sage: T = AbelianGroup([])
sage: T.is_cyclic()
True
sage: T = AbelianGroup(1, [0]); T
Multiplicative Abelian group isomorphic to Z
sage: T.is_cyclic()
True
sage: B = AbelianGroup([3,4,5])
sage: B.is_cyclic()
True
```

```
>>> from sage.all import *
>>> J = AbelianGroup([Integer(2),Integer(3)])
>>> J.gens_orders()
(2, 3)
>>> J.elementary_divisors()
(6,)
>>> J.is_cyclic()
True
>>> G = AbelianGroup([Integer(6)])
>>> G.gens_orders()
(6,)
>>> G.is_cyclic()
True
>>> H = AbelianGroup([Integer(2),Integer(2)])
>>> H.gens_orders()
(2, 2)
```

(continues on next page)

(continued from previous page)

```
>>> H.is_cyclic()
False
>>> H = AbelianGroup([Integer(2),Integer(4)])
>>> H.elementary_divisors()
(2, 4)
>>> H.is_cyclic()
False
>>> H.permutation_group().is_cyclic()                                        #␣
↪needs sage.groups
False
>>> T = AbelianGroup([])
>>> T.is_cyclic()
True
>>> T = AbelianGroup(Integer(1), [Integer(0)]); T
Multiplicative Abelian group isomorphic to Z
>>> T.is_cyclic()
True
>>> B = AbelianGroup([Integer(3),Integer(4),Integer(5)])
>>> B.is_cyclic()
True
```

is_isomorphic(*left*, *right*)

Check whether `left` and `right` are isomorphic.

INPUT:

- `right` – anything

OUTPUT: boolean; whether `left` and `right` are isomorphic as abelian groups

EXAMPLES:

```
sage: G1 = AbelianGroup([2,3,4,5])
sage: G2 = AbelianGroup([2,3,4,5,1])
sage: G1.is_isomorphic(G2)
True
```

```
>>> from sage.all import *
>>> G1 = AbelianGroup([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> G2 = AbelianGroup([Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(1)])
>>> G1.is_isomorphic(G2)
True
```

is_subgroup(*left*, *right*)

Test whether `left` is a subgroup of `right`.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,4,5])
sage: G.is_subgroup(G)
True

sage: H = G.subgroup([G.1])                                                   #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.gap  # optional - gap_package_polycyclic
sage: H.is_subgroup(G)                                                          #␣
↪needs sage.libs.gap  # optional - gap_package_polycyclic
True

sage: G.<a, b> = AbelianGroup(2)
sage: H.<c> = AbelianGroup(1)
sage: H < G
False
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> G.is_subgroup(G)
True

>>> H = G.subgroup([G.gen(1)])                                                  ␣
↪ # needs sage.libs.gap  # optional - gap_package_polycyclic
>>> H.is_subgroup(G)                                                          #␣
↪needs sage.libs.gap  # optional - gap_package_polycyclic
True

>>> G = AbelianGroup(Integer(2), names=('a', 'b',)); (a, b,) = G._first_
↪ngens(2)
>>> H = AbelianGroup(Integer(1), names=('c',)); (c,) = H._first_ngens(1)
>>> H < G
False
```

is_trivial()

Return whether the group is trivial.

A group is trivial if it has precisely one element.

EXAMPLES:

```
sage: AbelianGroup([2, 3]).is_trivial()
False
sage: AbelianGroup([1, 1]).is_trivial()
True
```

```
>>> from sage.all import *
>>> AbelianGroup([Integer(2), Integer(3)]).is_trivial()
False
>>> AbelianGroup([Integer(1), Integer(1)]).is_trivial()
True
```

list()

Return tuple of all elements of this group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3], names = "ab")
sage: G.list()
(1, b, b^2, a, a*b, a*b^2)
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(2),Integer(3)], names = "ab")
>>> G.list()
(1, b, b^2, a, a*b, a*b^2)
```

```
sage: G = AbelianGroup([]); G
Trivial Abelian group
sage: G.list()
(1,)
```

```
>>> from sage.all import *
>>> G = AbelianGroup([]); G
Trivial Abelian group
>>> G.list()
(1,)
```

ngens()

The number of free generators of the abelian group.

EXAMPLES:

```
sage: F = AbelianGroup(10000)
sage: F.ngens()
10000
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(10000))
>>> F.ngens()
10000
```

number_of_subgroups(*order=None*)

Return the number of subgroups of this group, possibly only of a specific order.

INPUT:

- `order` – (default: `None`) find the number of subgroups of this order; if `None`, this defaults to counting all subgroups

ALGORITHM:

An infinite group has infinitely many subgroups. All finite subgroups of any group are contained in the torsion subgroup, which for finitely generated abelian group is itself finite. Hence, we can assume the group is finite. A finite abelian group is isomorphic to a direct product of its Sylow subgroups, and so we can reduce the problem further to counting subgroups of finite abelian p-groups.

Assume a Sylow subgroup is a p-group of type λ, and using `q_subgroups_of_abelian_group()` sum the number of subgroups of type μ in an abelian p-group of type λ for all μ contained in λ.

EXAMPLES:

```
sage: AbelianGroup([2,0,0,3,0]).number_of_subgroups()
+Infinity

sage: # needs sage.combinat
sage: AbelianGroup([2,3]).number_of_subgroups()
```

(continues on next page)

(continued from previous page)

```
4
sage: AbelianGroup([2,4,8]).number_of_subgroups()
81
sage: AbelianGroup([2,4,8]).number_of_subgroups(order=4)
19
sage: AbelianGroup([10,15,25,12]).number_of_subgroups()
5760
sage: AbelianGroup([10,15,25,12]).number_of_subgroups(order=45000)
1
sage: AbelianGroup([10,15,25,12]).number_of_subgroups(order=14)
0
```

```
>>> from sage.all import *
>>> AbelianGroup([Integer(2),Integer(0),Integer(0),Integer(3),Integer(0)]).
↪number_of_subgroups()
+Infinity

>>> # needs sage.combinat
>>> AbelianGroup([Integer(2),Integer(3)]).number_of_subgroups()
4
>>> AbelianGroup([Integer(2),Integer(4),Integer(8)]).number_of_subgroups()
81
>>> AbelianGroup([Integer(2),Integer(4),Integer(8)]).number_of_
↪subgroups(order=Integer(4))
19
>>> AbelianGroup([Integer(10),Integer(15),Integer(25),Integer(12)]).number_of_
↪subgroups()
5760
>>> AbelianGroup([Integer(10),Integer(15),Integer(25),Integer(12)]).number_of_
↪subgroups(order=Integer(45000))
1
>>> AbelianGroup([Integer(10),Integer(15),Integer(25),Integer(12)]).number_of_
↪subgroups(order=Integer(14))
0
```

order()

Return the order of this group.

EXAMPLES:

```
sage: G = AbelianGroup(2, [2,3])
sage: G.order()
6
sage: G = AbelianGroup(3, [2,3,0])
sage: G.order()
+Infinity
```

```
>>> from sage.all import *
>>> G = AbelianGroup(Integer(2), [Integer(2),Integer(3)])
>>> G.order()
6
>>> G = AbelianGroup(Integer(3), [Integer(2),Integer(3),Integer(0)])
```

(continues on next page)

(continued from previous page)

```
>>> G.order()
+Infinity
```

permutation_group()

Return the permutation group isomorphic to this abelian group.

If the invariants are $q_1, \ldots, q_n$ then the generators of the permutation will be of order $q_1, \ldots, q_n$, respectively.

EXAMPLES:

```
sage: G = AbelianGroup(2,[2,3]); G
Multiplicative Abelian group isomorphic to C2 x C3
sage: G.permutation_group()                                              #␣
↪needs sage.groups
Permutation Group with generators [(1,2), (3,4,5)]
```

```
>>> from sage.all import *
>>> G = AbelianGroup(Integer(2),[Integer(2),Integer(3)]); G
Multiplicative Abelian group isomorphic to C2 x C3
>>> G.permutation_group()                                              #␣
↪needs sage.groups
Permutation Group with generators [(1,2), (3,4,5)]
```

random_element()

Return a random element of this group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,9])
sage: G.random_element().parent() is G
True
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(2),Integer(3),Integer(9)])
>>> G.random_element().parent() is G
True
```

subgroup(*gensH*, *names='f'*)

Create a subgroup of this group.

The "big" group must be defined using "named" generators.

INPUT:

- `gensH` – list of elements which are products of the generators of the ambient abelian group G = `self`

EXAMPLES:

```
sage: # needs sage.libs.gap  # optional - gap_package_polycyclic
sage: G.<a,b,c> = AbelianGroup(3, [2,3,4]); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
sage: H = G.subgroup([a*b,a]); H
Multiplicative Abelian subgroup isomorphic to C2 x C3 generated by {a*b, a}
sage: H < G
True
```

(continues on next page)

(continued from previous page)

```
sage: F = G.subgroup([a,b^2])
sage: F
Multiplicative Abelian subgroup isomorphic to C2 x C3 generated by {a, b^2}
sage: F.gens()
(a, b^2)
sage: F = AbelianGroup(5, [30,64,729], names=list("abcde"))
sage: a,b,c,d,e = F.gens()
sage: F.subgroup([a,b])
Multiplicative Abelian subgroup isomorphic to Z x Z generated by {a, b}
sage: F.subgroup([c,e])
Multiplicative Abelian subgroup isomorphic to C2 x C3 x C5 x C729
generated by {c, e}
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap  # optional - gap_package_polycyclic
>>> G = AbelianGroup(Integer(3), [Integer(2),Integer(3),Integer(4)], names=('a
↪', 'b', 'c',)); (a, b, c,) = G._first_ngens(3); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
>>> H = G.subgroup([a*b,a]); H
Multiplicative Abelian subgroup isomorphic to C2 x C3 generated by {a*b, a}
>>> H < G
True
>>> F = G.subgroup([a,b**Integer(2)])
>>> F
Multiplicative Abelian subgroup isomorphic to C2 x C3 generated by {a, b^2}
>>> F.gens()
(a, b^2)
>>> F = AbelianGroup(Integer(5), [Integer(30),Integer(64),Integer(729)],␣
↪names=list("abcde"))
>>> a,b,c,d,e = F.gens()
>>> F.subgroup([a,b])
Multiplicative Abelian subgroup isomorphic to Z x Z generated by {a, b}
>>> F.subgroup([c,e])
Multiplicative Abelian subgroup isomorphic to C2 x C3 x C5 x C729
generated by {c, e}
```

subgroup_reduced(*elts*, *verbose=False*)

Given a list of lists of integers (corresponding to elements of `self`), find a set of independent generators for the subgroup generated by these elements, and return the subgroup with these as generators, forgetting the original generators.

This is used by the `subgroups` routine.

An error will be raised if the elements given are not linearly independent over **Q**.

EXAMPLES:

```
sage: G = AbelianGroup([4,4])
sage: G.subgroup( [ G([1,0]), G([1,2]) ])                              #␣
↪needs sage.libs.gap  # optional - gap_package_polycyclic
Multiplicative Abelian subgroup isomorphic to C2 x C4
generated by {f0, f0*f1^2}
sage: AbelianGroup([4,4]).subgroup_reduced( [ [1,0], [1,2] ])          #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.gap  # optional - gap_package_polycyclic
Multiplicative Abelian subgroup isomorphic to C2 x C4
generated by {f0^2*f1^2, f0^3}
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(4),Integer(4)])
>>> G.subgroup( [ G([Integer(1),Integer(0)]), G([Integer(1),Integer(2)]) ])  ␣
↪                          # needs sage.libs.gap  # optional - gap_
↪package_polycyclic
Multiplicative Abelian subgroup isomorphic to C2 x C4
generated by {f0, f0*f1^2}
>>> AbelianGroup([Integer(4),Integer(4)]).subgroup_reduced( [ [Integer(1),
↪Integer(0)], [Integer(1),Integer(2)] ])            # needs sage.libs.gap␣
↪ # optional - gap_package_polycyclic
Multiplicative Abelian subgroup isomorphic to C2 x C4
generated by {f0^2*f1^2, f0^3}
```

subgroups(*check=False*)

Compute all the subgroups of this abelian group (which must be finite).

INPUT:

- `check` – boolean; if `True`, performs the same computation in GAP and checks that the number of subgroups generated is the same. (I don't know how to convert GAP's output back into Sage, so we don't actually compare the subgroups).

ALGORITHM:

If the group is cyclic, the problem is easy. Otherwise, write it as a direct product A x B, where B is cyclic. Compute the subgroups of A (by recursion).

Now, for every subgroup C of A x B, let G be its *projection onto* A and H its *intersection with* B. Then there is a well-defined homomorphism f: G -> B/H that sends a in G to the class mod H of b, where (a,b) is any element of C lifting a; and every subgroup C arises from a unique triple (G, H, f).

Todo

This is *many orders of magnitude* slower than Magma. Consider using the much faster method `number_of_subgroups()` in case you only need the number of subgroups, possibly of a specific order.

EXAMPLES:

```
sage: AbelianGroup([2,3]).subgroups()                                    #␣
↪needs sage.libs.gap  # optional - gap_package_polycyclic
[Multiplicative Abelian subgroup isomorphic to C2 x C3 generated by {f0*f1^2},
 Multiplicative Abelian subgroup isomorphic to C2 generated by {f0},
 Multiplicative Abelian subgroup isomorphic to C3 generated by {f1},
 Trivial Abelian subgroup]
sage: len(AbelianGroup([2,4,8]).subgroups())                             #␣
↪needs sage.libs.gap  # optional - gap_package_polycyclic
81
```

```
>>> from sage.all import *
>>> AbelianGroup([Integer(2),Integer(3)]).subgroups()                          ␣
↪              # needs sage.libs.gap  # optional - gap_package_polycyclic
[Multiplicative Abelian subgroup isomorphic to C2 x C3 generated by {f0*f1^2},
 Multiplicative Abelian subgroup isomorphic to C2 generated by {f0},
 Multiplicative Abelian subgroup isomorphic to C3 generated by {f1},
 Trivial Abelian subgroup]
>>> len(AbelianGroup([Integer(2),Integer(4),Integer(8)]).subgroups())          ␣
↪                       # needs sage.libs.gap  # optional - gap_package_
↪polycyclic
81
```

torsion_subgroup(*n=None*)

Return the n-torsion subgroup of this abelian group when n is given, and the torsion subgroup otherwise.

The [n-]torsion subgroup consists of all elements whose order is finite [and divides n].

EXAMPLES:

```
sage: # needs sage.libs.gap  # optional - gap_package_polycyclic
sage: G = AbelianGroup([2, 3])
sage: G.torsion_subgroup()
Multiplicative Abelian subgroup isomorphic to C2 x C3 generated
by {f0, f1}
sage: G = AbelianGroup([2, 0, 0, 3, 0])
sage: G.torsion_subgroup()
Multiplicative Abelian subgroup isomorphic to C2 x C3 generated
by {f0, f3}
sage: G = AbelianGroup([])
sage: G.torsion_subgroup()
Trivial Abelian subgroup
sage: G = AbelianGroup([0, 0])
sage: G.torsion_subgroup()
Trivial Abelian subgroup
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap  # optional - gap_package_polycyclic
>>> G = AbelianGroup([Integer(2), Integer(3)])
>>> G.torsion_subgroup()
Multiplicative Abelian subgroup isomorphic to C2 x C3 generated
by {f0, f1}
>>> G = AbelianGroup([Integer(2), Integer(0), Integer(0), Integer(3),␣
↪Integer(0)])
>>> G.torsion_subgroup()
Multiplicative Abelian subgroup isomorphic to C2 x C3 generated
by {f0, f3}
>>> G = AbelianGroup([])
>>> G.torsion_subgroup()
Trivial Abelian subgroup
>>> G = AbelianGroup([Integer(0), Integer(0)])
>>> G.torsion_subgroup()
Trivial Abelian subgroup
```

```
sage: G = AbelianGroup([2, 2*3, 2*3*5, 0, 2*3*5*7, 2*3*5*7*11])
sage: G.torsion_subgroup(5)                                                        #␣
↪needs sage.libs.gap  # optional - gap_package_polycyclic
Multiplicative Abelian subgroup isomorphic to C5 x C5 x C5
 generated by {f2^6, f4^42, f5^462}
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(2), Integer(2)*Integer(3),␣
↪Integer(2)*Integer(3)*Integer(5), Integer(0),␣
↪Integer(2)*Integer(3)*Integer(5)*Integer(7),␣
↪Integer(2)*Integer(3)*Integer(5)*Integer(7)*Integer(11)])
>>> G.torsion_subgroup(Integer(5))                                                 ␣
↪     # needs sage.libs.gap  # optional - gap_package_polycyclic
Multiplicative Abelian subgroup isomorphic to C5 x C5 x C5
 generated by {f2^6, f4^42, f5^462}
```

class sage.groups.abelian_gps.abelian_group.**AbelianGroup_subgroup**(*ambient*, *gens*, *names='f'*, *category=None*)

Bases: *AbelianGroup_class*

Subgroup subclass of AbelianGroup_class, so instance methods are inherited.

Todo

There should be a way to coerce an element of a subgroup into the ambient group.

ambient_group()

Return the ambient group related to self.

OUTPUT: a multiplicative Abelian group

EXAMPLES:

```
sage: # needs sage.libs.gap  # optional - gap_package_polycyclic
sage: G.<a,b,c> = AbelianGroup([2,3,4])
sage: H = G.subgroup([a, b^2])
sage: H.ambient_group() is G
True
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap  # optional - gap_package_polycyclic
>>> G = AbelianGroup([Integer(2),Integer(3),Integer(4)], names=('a', 'b', 'c',
↪)); (a, b, c,) = G._first_ngens(3)
>>> H = G.subgroup([a, b**Integer(2)])
>>> H.ambient_group() is G
True
```

equals(*left*, *right*)

Check whether left and right are the same (sub)group.

INPUT:

- right – anything

OUTPUT:

boolean; if `right` is a subgroup, test whether `left` and `right` are the same subset of the ambient group. If `right` is not a subgroup, test whether they are isomorphic groups, see `is_isomorphic()`.

EXAMPLES:

```
sage: # needs sage.libs.gap  # optional - gap_package_polycyclic
sage: G = AbelianGroup(3, [2,3,4], names='abc'); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
sage: a,b,c = G.gens()
sage: F = G.subgroup([a,b^2]); F
Multiplicative Abelian subgroup isomorphic to C2 x C3 generated by {a, b^2}
sage: F<G
True
sage: A = AbelianGroup(1, [6])
sage: A.subgroup(list(A.gens())) == A
True
sage: G.<a,b> = AbelianGroup(2)
sage: A = G.subgroup([a])
sage: B = G.subgroup([b])
sage: A.equals(B)
False
sage: A == B        # sames as A.equals(B)
False
sage: A.is_isomorphic(B)
True
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap  # optional - gap_package_polycyclic
>>> G = AbelianGroup(Integer(3), [Integer(2),Integer(3),Integer(4)], names=
↪'abc'); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
>>> a,b,c = G.gens()
>>> F = G.subgroup([a,b**Integer(2)]); F
Multiplicative Abelian subgroup isomorphic to C2 x C3 generated by {a, b^2}
>>> F<G
True
>>> A = AbelianGroup(Integer(1), [Integer(6)])
>>> A.subgroup(list(A.gens())) == A
True
>>> G = AbelianGroup(Integer(2), names=('a', 'b',)); (a, b,) = G._first_
↪ngens(2)
>>> A = G.subgroup([a])
>>> B = G.subgroup([b])
>>> A.equals(B)
False
>>> A == B        # sames as A.equals(B)
False
>>> A.is_isomorphic(B)
True
```

gen(n)

Return the n-th generator of this subgroup.

EXAMPLES:

```
sage: # needs sage.libs.gap  # optional - gap_package_polycyclic
sage: G.<a,b> = AbelianGroup(2)
sage: A = G.subgroup([a])
sage: A.gen(0)
a
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap  # optional - gap_package_polycyclic
>>> G = AbelianGroup(Integer(2), names=('a', 'b',)); (a, b,) = G._first_
↪ngens(2)
>>> A = G.subgroup([a])
>>> A.gen(Integer(0))
a
```

gens()

Return the generators for this subgroup.

OUTPUT: tuple of group elements generating the subgroup

EXAMPLES:

```
sage: # needs sage.libs.gap  # optional - gap_package_polycyclic
sage: G.<a,b> = AbelianGroup(2)
sage: A = G.subgroup([a])
sage: G.gens()
(a, b)
sage: A.gens()
(a,)
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap  # optional - gap_package_polycyclic
>>> G = AbelianGroup(Integer(2), names=('a', 'b',)); (a, b,) = G._first_
↪ngens(2)
>>> A = G.subgroup([a])
>>> G.gens()
(a, b)
>>> A.gens()
(a,)
```

sage.groups.abelian_gps.abelian_group.**is_AbelianGroup**(*x*)

Return `True` if `x` is an Abelian group.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group import is_AbelianGroup
sage: F = AbelianGroup(5,[5,5,7,8,9], names=list("abcde")); F
Multiplicative Abelian group isomorphic to C5 x C5 x C7 x C8 x C9
sage: is_AbelianGroup(F)
doctest:warning...
DeprecationWarning: the function is_AbelianGroup is deprecated;
use 'isinstance(..., AbelianGroup_class)' instead
See https://github.com/sagemath/sage/issues/37898 for details.
```

(continues on next page)

(continued from previous page)

```
True
sage: is_AbelianGroup(AbelianGroup(7, [3]*7))
True
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group import is_AbelianGroup
>>> F = AbelianGroup(Integer(5),[Integer(5),Integer(5),Integer(7),Integer(8),
↪Integer(9)], names=list("abcde")); F
Multiplicative Abelian group isomorphic to C5 x C5 x C7 x C8 x C9
>>> is_AbelianGroup(F)
doctest:warning...
DeprecationWarning: the function is_AbelianGroup is deprecated;
use 'isinstance(..., AbelianGroup_class)' instead
See https://github.com/sagemath/sage/issues/37898 for details.
True
>>> is_AbelianGroup(AbelianGroup(Integer(7), [Integer(3)]*Integer(7)))
True
```

sage.groups.abelian_gps.abelian_group.**word_problem**(*words*, *g*, *verbose=False*)

G and H are abelian, g in G, H is a subgroup of G generated by a list (words) of elements of G. If g is in H, return the expression for g as a word in the elements of (words).

The 'word problem' for a finite abelian group G boils down to the following matrix-vector analog of the Chinese remainder theorem.

Problem: Fix integers $1 < n_1 \leq n_2 \leq ... \leq n_k$ (indeed, these n_i will all be prime powers), fix a generating set $g_i = (a_{i1}, ..., a_{ik})$ (with $a_{ij} \in Z/n_jZ$), for $1 \leq i \leq \ell$, for the group G, and let $d = (d_1, ..., d_k)$ be an element of the direct product $Z/n_1Z \times ... \times Z/n_kZ$. Find, if they exist, integers $c_1, ..., c_\ell$ such that $c_1g_1 + ... + c_\ell g_\ell = d$. In other words, solve the equation $cA = d$ for $c \in Z^\ell$, where A is the matrix whose rows are the g_i's. Of course, it suffices to restrict the c_i's to the range $0 \leq c_i \leq N - 1$, where N denotes the least common multiple of the integers $n_1, ..., n_k$.

This function does not solve this directly, as perhaps it should. Rather (for both speed and as a model for a similar function valid for more general groups), it pushes it over to GAP, which has optimized (non-deterministic) algorithms for the word problem. Essentially, this function is a wrapper for the GAP function 'Factorization'.

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: G.<a,b,c> = AbelianGroup(3, [2,3,4]); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
sage: w = word_problem([a*b,a*c], b*c); w       # random
[[a*b, 1], [a*c, 1]]
sage: prod([x^i for x,i in w]) == b*c
True
sage: w = word_problem([a*c,c], a); w           # random
[[a*c, 1], [c, -1]]
sage: prod([x^i for x,i in w]) == a
True
sage: word_problem([a*c,c], a, verbose=True)    # random
a = (a*c)^1*(c)^-1
[[a*c, 1], [c, -1]]
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> G = AbelianGroup(Integer(3), [Integer(2),Integer(3),Integer(4)], names=('a',
↪'b', 'c',)); (a, b, c,) = G._first_ngens(3); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
>>> w = word_problem([a*b,a*c], b*c); w        # random
[[a*b, 1], [a*c, 1]]
>>> prod([x**i for x,i in w]) == b*c
True
>>> w = word_problem([a*c,c], a); w            # random
[[a*c, 1], [c, -1]]
>>> prod([x**i for x,i in w]) == a
True
>>> word_problem([a*c,c], a, verbose=True)     # random
a = (a*c)^1*(c)^-1
[[a*c, 1], [c, -1]]
```

```
sage: # needs sage.libs.gap
sage: A.<a,b,c,d,e> = AbelianGroup(5, [4, 5, 5, 7, 8])
sage: b1 = a^3*b*c*d^2*e^5
sage: b2 = a^2*b*c^2*d^3*e^3
sage: b3 = a^7*b^3*c^5*d^4*e^4
sage: b4 = a^3*b^2*c^2*d^3*e^5
sage: b5 = a^2*b^4*c^2*d^4*e^5
sage: w = word_problem([b1,b2,b3,b4,b5], e); w  # random
[[a^3*b*c*d^2*e^5, 1], [a^2*b*c^2*d^3*e^3, 1],
 [a^3*b^3*d^4*e^4, 3], [a^2*b^4*c^2*d^4*e^5, 1]]
sage: prod([x^i for x,i in w]) == e
True
sage: word_problem([a,b,c,d,e], e)
[[e, 1]]
sage: word_problem([a,b,c,d,e], b)
[[b, 1]]
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> A = AbelianGroup(Integer(5), [Integer(4), Integer(5), Integer(5), Integer(7),␣
↪Integer(8)], names=('a', 'b', 'c', 'd', 'e',)); (a, b, c, d, e,) = A._first_
↪ngens(5)
>>> b1 = a**Integer(3)*b*c*d**Integer(2)*e**Integer(5)
>>> b2 = a**Integer(2)*b*c**Integer(2)*d**Integer(3)*e**Integer(3)
>>> b3 = a**Integer(7)*b**Integer(3)*c**Integer(5)*d**Integer(4)*e**Integer(4)
>>> b4 = a**Integer(3)*b**Integer(2)*c**Integer(2)*d**Integer(3)*e**Integer(5)
>>> b5 = a**Integer(2)*b**Integer(4)*c**Integer(2)*d**Integer(4)*e**Integer(5)
>>> w = word_problem([b1,b2,b3,b4,b5], e); w  # random
[[a^3*b*c*d^2*e^5, 1], [a^2*b*c^2*d^3*e^3, 1],
 [a^3*b^3*d^4*e^4, 3], [a^2*b^4*c^2*d^4*e^5, 1]]
>>> prod([x**i for x,i in w]) == e
True
>>> word_problem([a,b,c,d,e], e)
[[e, 1]]
>>> word_problem([a,b,c,d,e], b)
```

(continues on next page)

(continued from previous page)

```
[[b, 1]]
```

Warning

1. Might have unpleasant effect when the word problem cannot be solved.
2. Uses permutation groups, so may be slow when group is large. The instance method word_problem of the class AbelianGroupElement is implemented differently (wrapping GAP's 'EpimorphismFromFreeGroup' and 'PreImagesRepresentative') and may be faster.

26.2 Finitely generated abelian groups with GAP.

This module provides a python wrapper for abelian groups in GAP.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: AbelianGroupGap([3,5])
Abelian group with gap, generator orders (3, 5)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> AbelianGroupGap([Integer(3),Integer(5)])
Abelian group with gap, generator orders (3, 5)
```

For infinite abelian groups we use the GAP package `Polycyclic`:

```
sage: AbelianGroupGap([3,0])    # optional - gap_package_polycyclic
Abelian group with gap, generator orders (3, 0)
```

```
>>> from sage.all import *
>>> AbelianGroupGap([Integer(3),Integer(0)])    # optional - gap_package_polycyclic
Abelian group with gap, generator orders (3, 0)
```

AUTHORS:

- Simon Brandhorst (2018-01-17): initial version

class sage.groups.abelian_gps.abelian_group_gap.**AbelianGroupElement_gap**(*parent*, *x*, *check=True*)

Bases: *ElementLibGAP*

An element of an abelian group via libgap.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([3,6])
sage: G.gens()
(f1, f2)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(3),Integer(6)])
>>> G.gens()
(f1, f2)
```

exponents()

Return the tuple of exponents of this element.

OUTPUT: tuple of integers

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([4,7,9])
sage: gens = G.gens()
sage: g = gens[0]^2 * gens[1]^4 * gens[2]^8
sage: g.exponents()
(2, 4, 8)
sage: S = G.subgroup(G.gens()[:1])
sage: s = S.gens()[0]
sage: s
f1
sage: s.exponents()
(1,)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(4),Integer(7),Integer(9)])
>>> gens = G.gens()
>>> g = gens[Integer(0)]**Integer(2) * gens[Integer(1)]**Integer(4) *␣
↪gens[Integer(2)]**Integer(8)
>>> g.exponents()
(2, 4, 8)
>>> S = G.subgroup(G.gens()[:Integer(1)])
>>> s = S.gens()[Integer(0)]
>>> s
f1
>>> s.exponents()
(1,)
```

It can handle quite large groups too:

```
sage: G = AbelianGroupGap([2^10, 5^10])
sage: f1, f2 = G.gens()
sage: g = f1^123*f2^789
sage: g.exponents()
(123, 789)
```

```
>>> from sage.all import *
>>> G = AbelianGroupGap([Integer(2)**Integer(10), Integer(5)**Integer(10)])
>>> f1, f2 = G.gens()
>>> g = f1**Integer(123)*f2**Integer(789)
```

(continues on next page)

(continued from previous page)

```
>>> g.exponents()
(123, 789)
```

Warning: Crashes for very large groups.

Todo: Make exponents work for very large groups. This could be done by using Pcgs in gap.

order()

Return the order of this element.

OUTPUT: integer or infinity

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([4])
sage: g = G.gens()[0]
sage: g.order()
4
sage: G = AbelianGroupGap([0])          # optional - gap_package_polycyclic
sage: g = G.gens()[0]                   # optional - gap_package_polycyclic
sage: g.order()                         # optional - gap_package_polycyclic
+Infinity
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(4)])
>>> g = G.gens()[Integer(0)]
>>> g.order()
4
>>> G = AbelianGroupGap([Integer(0)])          # optional - gap_package_
↪polycyclic
>>> g = G.gens()[Integer(0)]                   # optional - gap_package_
↪polycyclic
>>> g.order()                         # optional - gap_package_polycyclic
+Infinity
```

class sage.groups.abelian_gps.abelian_group_gap.**AbelianGroupElement_polycyclic**(*parent*, *x*, *check=True*)

Bases: `AbelianGroupElement_gap`

An element of an abelian group using the GAP package `Polycyclic`.

exponents()

Return the tuple of exponents of `self`.

OUTPUT: tuple of integers

EXAMPLES:

```
sage: # optional - gap_package_polycyclic
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([4,7,0])
sage: gens = G.gens()
sage: g = gens[0]^2 * gens[1]^4 * gens[2]^8
sage: g.exponents()
(2, 4, 8)
```

```
>>> from sage.all import *
>>> # optional - gap_package_polycyclic
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(4),Integer(7),Integer(0)])
>>> gens = G.gens()
>>> g = gens[Integer(0)]**Integer(2) * gens[Integer(1)]**Integer(4) *␣
↪gens[Integer(2)]**Integer(8)
>>> g.exponents()
(2, 4, 8)
```

Efficiently handles very large groups:

```
sage: # optional - gap_package_polycyclic
sage: G = AbelianGroupGap([2^30,5^30,0])
sage: f1, f2, f3 = G.gens()
sage: (f1^12345*f2^123456789).exponents()
(12345, 123456789, 0)
```

```
>>> from sage.all import *
>>> # optional - gap_package_polycyclic
>>> G = AbelianGroupGap([Integer(2)**Integer(30),Integer(5)**Integer(30),
↪Integer(0)])
>>> f1, f2, f3 = G.gens()
>>> (f1**Integer(12345)*f2**Integer(123456789)).exponents()
(12345, 123456789, 0)
```

class sage.groups.abelian_gps.abelian_group_gap.**AbelianGroupGap**(*generator_orders*)

Bases: *AbelianGroup_gap*

Abelian groups implemented using GAP.

INPUT:

- generator_orders – list of nonnegative integers where 0 gives a factor isomorphic to $\mathbf{Z}$

OUTPUT: an abelian group

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: AbelianGroupGap([3,6])
Abelian group with gap, generator orders (3, 6)
sage: AbelianGroupGap([3,6,5])
Abelian group with gap, generator orders (3, 6, 5)
sage: AbelianGroupGap([3,6,0])      # optional - gap_package_polycyclic
Abelian group with gap, generator orders (3, 6, 0)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> AbelianGroupGap([Integer(3),Integer(6)])
Abelian group with gap, generator orders (3, 6)
>>> AbelianGroupGap([Integer(3),Integer(6),Integer(5)])
Abelian group with gap, generator orders (3, 6, 5)
>>> AbelianGroupGap([Integer(3),Integer(6),Integer(0)])       # optional - gap_
↪package_polycyclic
Abelian group with gap, generator orders (3, 6, 0)
```

Warning

Needs the GAP package `Polycyclic` in case the group is infinite.

class sage.groups.abelian_gps.abelian_group_gap.**AbelianGroupQuotient_gap**(G, N)

Bases: *AbelianGroup_gap*

Quotients of abelian groups by a subgroup.

Note

Do not call this directly. Instead use `quotient()`.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: A = AbelianGroupGap([4,3])
sage: N = A.subgroup([A.gen(0)^2])
sage: Q1 = A.quotient(N)
sage: Q1
Quotient abelian group with generator orders (2, 3)
sage: Q2 = Q1.quotient(Q1.subgroup(Q1.gens()[:1]))
sage: Q2
Quotient abelian group with generator orders (1, 3)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> A = AbelianGroupGap([Integer(4),Integer(3)])
>>> N = A.subgroup([A.gen(Integer(0))**Integer(2)])
>>> Q1 = A.quotient(N)
>>> Q1
Quotient abelian group with generator orders (2, 3)
>>> Q2 = Q1.quotient(Q1.subgroup(Q1.gens()[:Integer(1)]))
>>> Q2
Quotient abelian group with generator orders (1, 3)
```

cover()

Return the covering group of this quotient group.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,4,5])
sage: gen = G.gens()[:2]
sage: S = G.subgroup(gen)
sage: Q = G.quotient(S)
sage: Q.cover() is G
True
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> gen = G.gens()[:Integer(2)]
>>> S = G.subgroup(gen)
>>> Q = G.quotient(S)
>>> Q.cover() is G
True
```

lift(*x*)

Lift an element to the cover.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: A = AbelianGroupGap([4])
sage: N = A.subgroup([A.gen(0)^2])
sage: Q = A.quotient(N)
sage: Q.lift(Q.0)
f1
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> A = AbelianGroupGap([Integer(4)])
>>> N = A.subgroup([A.gen(Integer(0))**Integer(2)])
>>> Q = A.quotient(N)
>>> Q.lift(Q.gen(0))
f1
```

natural_homomorphism()

Return the defining homomorphism into `self`.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: A = AbelianGroupGap([4])
sage: N = A.subgroup([A.gen(0)^2])
sage: Q = A.quotient(N)
sage: Q.natural_homomorphism()
Group morphism:
From: Abelian group with gap, generator orders (4,)
To:   Quotient abelian group with generator orders (2,)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
```

(continues on next page)

(continued from previous page)

```
>>> A = AbelianGroupGap([Integer(4)])
>>> N = A.subgroup([A.gen(Integer(0))**Integer(2)])
>>> Q = A.quotient(N)
>>> Q.natural_homomorphism()
Group morphism:
From: Abelian group with gap, generator orders (4,)
To:   Quotient abelian group with generator orders (2,)
```

relations()

Return the relations of this quotient group.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,4,5])
sage: gen = G.gens()[:2]
sage: S = G.subgroup(gen)
sage: Q = G.quotient(S)
sage: Q.relations() is S
True
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> gen = G.gens()[:Integer(2)]
>>> S = G.subgroup(gen)
>>> Q = G.quotient(S)
>>> Q.relations() is S
True
```

class sage.groups.abelian_gps.abelian_group_gap.**AbelianGroupSubgroup_gap**(*ambient*, *gens*)

Bases: *AbelianGroup_gap*

Subgroups of abelian groups with GAP.

INPUT:

- `ambient` – the ambient group
- `gens` – generators of the subgroup

Note

Do not construct this class directly. Instead use `subgroup()`.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,4,5])
sage: gen = G.gens()[:2]
sage: S = G.subgroup(gen)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> gen = G.gens()[:Integer(2)]
>>> S = G.subgroup(gen)
```

lift(*x*)

Coerce to the ambient group.

The terminology comes from the category framework and the more general notion of a subquotient.

INPUT:

- x – an element of this subgroup

OUTPUT: the corresponding element of the ambient group

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([4])
sage: g = G.gen(0)
sage: H = G.subgroup([g^2])
sage: h = H.gen(0); h
f2
sage: h.parent()
Subgroup of Abelian group with gap, generator orders (4,) generated by (f2,)
sage: H.lift(h)
f2
sage: H.lift(h).parent()
Abelian group with gap, generator orders (4,)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(4)])
>>> g = G.gen(Integer(0))
>>> H = G.subgroup([g**Integer(2)])
>>> h = H.gen(Integer(0)); h
f2
>>> h.parent()
Subgroup of Abelian group with gap, generator orders (4,) generated by (f2,)
>>> H.lift(h)
f2
>>> H.lift(h).parent()
Abelian group with gap, generator orders (4,)
```

retract(*x*)

Convert an element of the ambient group into this subgroup.

The terminology comes from the category framework and the more general notion of a subquotient.

INPUT:

- x – an element of the ambient group that actually lies in this subgroup

OUTPUT: the corresponding element of this subgroup

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([4])
sage: g = G.gen(0)
sage: H = G.subgroup([g^2])
sage: H.retract(g^2)
f2
sage: H.retract(g^2).parent()
Subgroup of Abelian group with gap, generator orders (4,) generated by (f2,)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(4)])
>>> g = G.gen(Integer(0))
>>> H = G.subgroup([g**Integer(2)])
>>> H.retract(g**Integer(2))
f2
>>> H.retract(g**Integer(2)).parent()
Subgroup of Abelian group with gap, generator orders (4,) generated by (f2,)
```

class `sage.groups.abelian_gps.abelian_group_gap.`**`AbelianGroup_gap`**(*G*, *category*, *ambient=None*)

Bases: `UniqueRepresentation`, *`GroupMixinLibGAP`*, *`ParentLibGAP`*, *`AbelianGroup`*

Finitely generated abelian groups implemented in GAP.

Needs the gap package `Polycyclic` in case the group is infinite.

INPUT:

- `G` – a GAP group
- `category` – a category
- `ambient` – (optional) an *`AbelianGroupGap`*

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([3, 2, 5])
sage: G
Abelian group with gap, generator orders (3, 2, 5)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(3), Integer(2), Integer(5)])
>>> G
Abelian group with gap, generator orders (3, 2, 5)
```

`Element`

alias of *`AbelianGroupElement_gap`*

`all_subgroups`()

Return the list of all subgroups of this group.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2, 3])
sage: G.all_subgroups()
[Subgroup of Abelian group with gap, generator orders (2, 3) generated by (),
 Subgroup of Abelian group with gap, generator orders (2, 3) generated by (f1,
↪),
 Subgroup of Abelian group with gap, generator orders (2, 3) generated by (f2,
↪),
 Subgroup of Abelian group with gap, generator orders (2, 3) generated by (f2,
↪ f1)]
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2), Integer(3)])
>>> G.all_subgroups()
[Subgroup of Abelian group with gap, generator orders (2, 3) generated by (),
 Subgroup of Abelian group with gap, generator orders (2, 3) generated by (f1,
↪),
 Subgroup of Abelian group with gap, generator orders (2, 3) generated by (f2,
↪),
 Subgroup of Abelian group with gap, generator orders (2, 3) generated by (f2,
↪ f1)]
```

aut()

Return the group of automorphisms of `self`.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2, 3])
sage: G.aut()
Full group of automorphisms of Abelian group with gap, generator orders (2, 3)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2), Integer(3)])
>>> G.aut()
Full group of automorphisms of Abelian group with gap, generator orders (2, 3)
```

automorphism_group()

Return the group of automorphisms of `self`.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2, 3])
sage: G.aut()
Full group of automorphisms of Abelian group with gap, generator orders (2, 3)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2), Integer(3)])
```

(continues on next page)

(continued from previous page)

```
>>> G.aut()
Full group of automorphisms of Abelian group with gap, generator orders (2, 3)
```

elementary_divisors()

Return the elementary divisors of this group.

See `sage.groups.abelian_gps.abelian_group_gap.elementary_divisors()`.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,4,5])
sage: G.elementary_divisors()
(2, 60)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> G.elementary_divisors()
(2, 60)
```

exponent()

Return the exponent of this abelian group.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,7])
sage: G
Abelian group with gap, generator orders (2, 3, 7)
sage: G = AbelianGroupGap([2,4,6])
sage: G
Abelian group with gap, generator orders (2, 4, 6)
sage: G.exponent()
12
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2),Integer(3),Integer(7)])
>>> G
Abelian group with gap, generator orders (2, 3, 7)
>>> G = AbelianGroupGap([Integer(2),Integer(4),Integer(6)])
>>> G
Abelian group with gap, generator orders (2, 4, 6)
>>> G.exponent()
12
```

gens_orders()

Return the orders of the generators.

Use *elementary_divisors()* if you are looking for an invariant of the group.

OUTPUT: tuple of integers

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: Z2xZ3 = AbelianGroupGap([2,3])
sage: Z2xZ3.gens_orders()
(2, 3)
sage: Z2xZ3.elementary_divisors()
(6,)
sage: Z6 = AbelianGroupGap([6])
sage: Z6.gens_orders()
(6,)
sage: Z6.elementary_divisors()
(6,)
sage: Z2xZ3.is_isomorphic(Z6)
True
sage: Z2xZ3 is Z6
False
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> Z2xZ3 = AbelianGroupGap([Integer(2),Integer(3)])
>>> Z2xZ3.gens_orders()
(2, 3)
>>> Z2xZ3.elementary_divisors()
(6,)
>>> Z6 = AbelianGroupGap([Integer(6)])
>>> Z6.gens_orders()
(6,)
>>> Z6.elementary_divisors()
(6,)
>>> Z2xZ3.is_isomorphic(Z6)
True
>>> Z2xZ3 is Z6
False
```

identity()

Return the identity element of this group.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([4,10])
sage: G.identity()
1
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(4),Integer(10)])
>>> G.identity()
1
```

is_subgroup_of(G)

Return if `self` is a subgroup of `G` considered in the same ambient group.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,4,5])
sage: gen = G.gens()[:2]
sage: S1 = G.subgroup(gen)
sage: S1.is_subgroup_of(G)
True
sage: S2 = G.subgroup(G.gens()[1:])
sage: S2.is_subgroup_of(S1)
False
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> gen = G.gens()[:Integer(2)]
>>> S1 = G.subgroup(gen)
>>> S1.is_subgroup_of(G)
True
>>> S2 = G.subgroup(G.gens()[Integer(1):])
>>> S2.is_subgroup_of(S1)
False
```

is_trivial()

Return `True` if this group is the trivial group.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([])
sage: G
Abelian group with gap, generator orders ()
sage: G.is_trivial()
True
sage: AbelianGroupGap([1]).is_trivial()
True
sage: AbelianGroupGap([1,1,1]).is_trivial()
True
sage: AbelianGroupGap([2]).is_trivial()
False
sage: AbelianGroupGap([2,1]).is_trivial()
False
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([])
>>> G
Abelian group with gap, generator orders ()
>>> G.is_trivial()
True
>>> AbelianGroupGap([Integer(1)]).is_trivial()
True
>>> AbelianGroupGap([Integer(1),Integer(1),Integer(1)]).is_trivial()
True
>>> AbelianGroupGap([Integer(2)]).is_trivial()
```

(continues on next page)

(continued from previous page)

```
False
>>> AbelianGroupGap([Integer(2),Integer(1)]).is_trivial()
False
```

quotient (*N*)

Return the quotient of this group by the normal subgroup N.

INPUT:

- `N` – a subgroup
- `check` – boolean (default: `True`); check if N is normal

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: A = AbelianGroupGap([2,3,4,5])
sage: S = A.subgroup(A.gens()[:1])
sage: A.quotient(S)
Quotient abelian group with generator orders (1, 3, 4, 5)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> A = AbelianGroupGap([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> S = A.subgroup(A.gens()[:Integer(1)])
>>> A.quotient(S)
Quotient abelian group with generator orders (1, 3, 4, 5)
```

subgroup (*gens*)

Return the subgroup of this group generated by `gens`.

INPUT:

- `gens` – list of elements coercible into this group

OUTPUT: a subgroup

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,4,5])
sage: gen = G.gens()[:2]
sage: S = G.subgroup(gen)
sage: S
Subgroup of Abelian group with gap, generator orders (2, 3, 4, 5)
 generated by (f1, f2)
sage: g = G.an_element()
sage: s = S.an_element()
sage: g * s
f2^2*f3*f5

sage: # optional - gap_package_polycyclic
sage: G = AbelianGroupGap([3,4,0,2])
sage: gen = G.gens()[:2]
sage: S = G.subgroup(gen)
sage: g = G.an_element()
```

(continues on next page)

(continued from previous page)

```
sage: s = S.an_element()
sage: g * s
g1^2*g2^2*g3*g4
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> gen = G.gens()[:Integer(2)]
>>> S = G.subgroup(gen)
>>> S
Subgroup of Abelian group with gap, generator orders (2, 3, 4, 5)
 generated by (f1, f2)
>>> g = G.an_element()
>>> s = S.an_element()
>>> g * s
f2^2*f3*f5

>>> # optional - gap_package_polycyclic
>>> G = AbelianGroupGap([Integer(3),Integer(4),Integer(0),Integer(2)])
>>> gen = G.gens()[:Integer(2)]
>>> S = G.subgroup(gen)
>>> g = G.an_element()
>>> s = S.an_element()
>>> g * s
g1^2*g2^2*g3*g4
```

26.3 Automorphisms of abelian groups

This implements groups of automorphisms of abelian groups.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,6])
sage: autG = G.aut()
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2),Integer(6)])
>>> autG = G.aut()
```

Automorphisms act on the elements of the domain:

```
sage: g = G.an_element()
sage: f = autG.an_element()
sage: f
Pcgs([ f1, f2, f3 ]) -> [ f1, f1*f2*f3^2, f3^2 ]
sage: (g, f(g))
(f1*f2, f2*f3^2)
```

```
>>> from sage.all import *
>>> g = G.an_element()
>>> f = autG.an_element()
>>> f
Pcgs([ f1, f2, f3 ]) -> [ f1, f1*f2*f3^2, f3^2 ]
>>> (g, f(g))
(f1*f2, f2*f3^2)
```

Or anything coercible into its domain:

```
sage: A = AbelianGroup([2,6])
sage: a = A.an_element()
sage: (a, f(a))
(f0*f1, f2*f3^2)
sage: A = AdditiveAbelianGroup([2,6])
sage: a = A.an_element()
sage: (a, f(a))
((1, 0), f1)
sage: f((1,1))
f2*f3^2
```

```
>>> from sage.all import *
>>> A = AbelianGroup([Integer(2),Integer(6)])
>>> a = A.an_element()
>>> (a, f(a))
(f0*f1, f2*f3^2)
>>> A = AdditiveAbelianGroup([Integer(2),Integer(6)])
>>> a = A.an_element()
>>> (a, f(a))
((1, 0), f1)
>>> f((Integer(1),Integer(1)))
f2*f3^2
```

We can compute conjugacy classes:

```
sage: autG.conjugacy_classes_representatives()
(1,
 Pcgs([ f1, f2, f3 ]) -> [ f2*f3, f1*f2, f3 ],
 Pcgs([ f1, f2, f3 ]) -> [ f1*f2*f3, f2*f3^2, f3^2 ],
 [ f3^2, f1*f2*f3, f1 ] -> [ f3^2, f1, f1*f2*f3 ],
 Pcgs([ f1, f2, f3 ]) -> [ f2*f3, f1*f2*f3^2, f3^2 ],
 [ f1*f2*f3, f1, f3^2 ] -> [ f1*f2*f3, f1, f3 ])
```

```
>>> from sage.all import *
>>> autG.conjugacy_classes_representatives()
(1,
 Pcgs([ f1, f2, f3 ]) -> [ f2*f3, f1*f2, f3 ],
 Pcgs([ f1, f2, f3 ]) -> [ f1*f2*f3, f2*f3^2, f3^2 ],
 [ f3^2, f1*f2*f3, f1 ] -> [ f3^2, f1, f1*f2*f3 ],
 Pcgs([ f1, f2, f3 ]) -> [ f2*f3, f1*f2*f3^2, f3^2 ],
 [ f1*f2*f3, f1, f3^2 ] -> [ f1*f2*f3, f1, f3 ])
```

the group order:

```
sage: autG.order()
12
```

```
>>> from sage.all import *
>>> autG.order()
12
```

or create subgroups and do the same for them:

```
sage: S = autG.subgroup(autG.gens()[:1])
sage: S
Subgroup of automorphisms of Abelian group with gap, generator orders (2, 6)
generated by 1 automorphisms
```

```
>>> from sage.all import *
>>> S = autG.subgroup(autG.gens()[:Integer(1)])
>>> S
Subgroup of automorphisms of Abelian group with gap, generator orders (2, 6)
generated by 1 automorphisms
```

Only automorphism groups of finite abelian groups are supported:

```
sage: G = AbelianGroupGap([0,2])         # optional - gap_package_polycyclic
sage: autG = G.aut()                     # optional - gap_package_polycyclic
Traceback (most recent call last):
...
ValueError: only finite abelian groups are supported
```

```
>>> from sage.all import *
>>> G = AbelianGroupGap([Integer(0),Integer(2)])         # optional - gap_package_
↪polycyclic
>>> autG = G.aut()                     # optional - gap_package_polycyclic
Traceback (most recent call last):
...
ValueError: only finite abelian groups are supported
```

AUTHORS:

- Simon Brandhorst (2018-02-17): initial version

class sage.groups.abelian_gps.abelian_aut.**AbelianGroupAutomorphism**(*parent*, *x*, *check=True*)

Bases: *ElementLibGAP*

Automorphisms of abelian groups with gap.

INPUT:

- `x` – a libgap element
- `parent` – the parent *AbelianGroupAutomorphismGroup_gap*
- `check` – boolean (default: `True`); checks if `x` is an element of the group

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,4,5])
sage: f = G.aut().an_element()
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> f = G.aut().an_element()
```

matrix()

Return the matrix defining `self`.

The i-th row is the exponent vector of the image of the i-th generator.

OUTPUT: a square matrix over the integers

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,4])
sage: f = G.aut().an_element()
sage: f
Pcgs([ f1, f2, f3, f4 ]) -> [ f1*f4, f2^2, f1*f3, f4 ]
sage: f.matrix()
[1 0 2]
[0 2 0]
[1 0 1]
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2),Integer(3),Integer(4)])
>>> f = G.aut().an_element()
>>> f
Pcgs([ f1, f2, f3, f4 ]) -> [ f1*f4, f2^2, f1*f3, f4 ]
>>> f.matrix()
[1 0 2]
[0 2 0]
[1 0 1]
```

Compare with the exponents of the images:

```
sage: f(G.gens()[0]).exponents()
(1, 0, 2)
sage: f(G.gens()[1]).exponents()
(0, 2, 0)
sage: f(G.gens()[2]).exponents()
(1, 0, 1)
```

```
>>> from sage.all import *
>>> f(G.gens()[Integer(0)]).exponents()
(1, 0, 2)
>>> f(G.gens()[Integer(1)]).exponents()
(0, 2, 0)
```

(continues on next page)

(continued from previous page)

```
>>> f(G.gens()[Integer(2)]).exponents()
(1, 0, 1)
```

class sage.groups.abelian_gps.abelian_aut.**AbelianGroupAutomorphismGroup**(*AbelianGroupGap*)

Bases: *AbelianGroupAutomorphismGroup_gap*

The full automorphism group of a finite abelian group.

INPUT:

- AbelianGroupGap – an instance of *AbelianGroup_gap*

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: from sage.groups.abelian_gps.abelian_aut import␣
↪AbelianGroupAutomorphismGroup
sage: G = AbelianGroupGap([2,3,4,5])
sage: aut = G.aut()
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> from sage.groups.abelian_gps.abelian_aut import AbelianGroupAutomorphismGroup
>>> G = AbelianGroupGap([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> aut = G.aut()
```

Equivalently:

```
sage: aut1 = AbelianGroupAutomorphismGroup(G)
sage: aut is aut1
True
```

```
>>> from sage.all import *
>>> aut1 = AbelianGroupAutomorphismGroup(G)
>>> aut is aut1
True
```

Element

alias of *AbelianGroupAutomorphism*

class sage.groups.abelian_gps.abelian_aut.**AbelianGroupAutomorphismGroup_gap**(*domain*, *gap_group*, *category*, *ambient=None*)

Bases: CachedRepresentation, *GroupMixinLibGAP*, *Group*, *ParentLibGAP*

Base class for groups of automorphisms of abelian groups.

Do not construct this directly.

INPUT:

- domain – *AbelianGroup_gap*
- libgap_parent – the libgap element that is the parent in GAP
- category – a category

- ambient – an instance of a derived class of *ParentLibGAP* or None (default); the ambient group if libgap_parent has been defined as a subgroup

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: from sage.groups.abelian_gps.abelian_aut import␣
↪AbelianGroupAutomorphismGroup_gap
sage: domain = AbelianGroupGap([2,3,4,5])
sage: aut = domain.gap().AutomorphismGroupAbelianGroup()
sage: AbelianGroupAutomorphismGroup_gap(domain, aut, Groups().Finite())
<group with 6 generators>
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> from sage.groups.abelian_gps.abelian_aut import AbelianGroupAutomorphismGroup_
↪gap
>>> domain = AbelianGroupGap([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> aut = domain.gap().AutomorphismGroupAbelianGroup()
>>> AbelianGroupAutomorphismGroup_gap(domain, aut, Groups().Finite())
<group with 6 generators>
```

Element

alias of *AbelianGroupAutomorphism*

covering_matrix_ring()

Return the covering matrix ring of this group.

This is the ring of $n \times n$ matrices over $\mathbf{Z}$ where n is the number of (independent) generators.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,4,5])
sage: aut = G.aut()
sage: aut.covering_matrix_ring()
Full MatrixSpace of 4 by 4 dense matrices over Integer Ring
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> aut = G.aut()
>>> aut.covering_matrix_ring()
Full MatrixSpace of 4 by 4 dense matrices over Integer Ring
```

domain()

Return the domain of this group of automorphisms.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,4,5])
sage: aut = G.aut()
sage: aut.domain()
Abelian group with gap, generator orders (2, 3, 4, 5)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> aut = G.aut()
>>> aut.domain()
Abelian group with gap, generator orders (2, 3, 4, 5)
```

is_subgroup_of(*G*)

Return if `self` is a subgroup of `G` considered in the same ambient group.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,4,5])
sage: aut = G.aut()
sage: gen = aut.gens()
sage: S1 = aut.subgroup(gen[:2])
sage: S1.is_subgroup_of(aut)
True
sage: S2 = aut.subgroup(aut.gens()[1:])
sage: S2.is_subgroup_of(S1)
False
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> G = AbelianGroupGap([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> aut = G.aut()
>>> gen = aut.gens()
>>> S1 = aut.subgroup(gen[:Integer(2)])
>>> S1.is_subgroup_of(aut)
True
>>> S2 = aut.subgroup(aut.gens()[Integer(1):])
>>> S2.is_subgroup_of(S1)
False
```

class sage.groups.abelian_gps.abelian_aut.**AbelianGroupAutomorphismGroup_subgroup**(*ambient*, *generators*)

Bases: *AbelianGroupAutomorphismGroup_gap*

Groups of automorphisms of abelian groups.

They are subgroups of the full automorphism group.

Note

Do not construct this class directly; instead use *sage.groups.abelian_gps.abelian_group_gap.AbelianGroup_gap.subgroup()*.

INPUT:

- `ambient` – the ambient group

- `generators` – tuple of gap elements of the ambient group

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: from sage.groups.abelian_gps.abelian_aut import
↪AbelianGroupAutomorphismGroup_subgroup
sage: G = AbelianGroupGap([2,3,4,5])
sage: aut = G.aut()
sage: gen = aut.gens()
sage: AbelianGroupAutomorphismGroup_subgroup(aut, gen)
Subgroup of automorphisms of Abelian group with gap, generator orders (2, 3, 4, 5)
generated by 6 automorphisms
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
>>> from sage.groups.abelian_gps.abelian_aut import AbelianGroupAutomorphismGroup_
↪subgroup
>>> G = AbelianGroupGap([Integer(2),Integer(3),Integer(4),Integer(5)])
>>> aut = G.aut()
>>> gen = aut.gens()
>>> AbelianGroupAutomorphismGroup_subgroup(aut, gen)
Subgroup of automorphisms of Abelian group with gap, generator orders (2, 3, 4, 5)
generated by 6 automorphisms
```

Element

alias of *AbelianGroupAutomorphism*

26.4 Multiplicative Abelian Groups With Values

Often, one ends up with a set that forms an Abelian group. It would be nice if one could return an Abelian group class to encapsulate the data. However, *AbelianGroup()* is an abstract Abelian group defined by generators and relations. This module implements *AbelianGroupWithValues* that allows the group elements to be decorated with values.

An example where this module is used is the unit group of a number field, see `sage.rings.number_field.unit_group`. The units form a finitely generated Abelian group. We can think of the elements either as abstract Abelian group elements or as particular numbers in the number field. The *AbelianGroupWithValues()* keeps track of these associated values.

Warning

Really, this requires a group homomorphism from the abstract Abelian group to the set of values. This is only checked if you pass the `check=True` option to *AbelianGroupWithValues()*.

EXAMPLES:

Here is $\mathbf{Z}_6$ with value -1 assigned to the generator:

```
sage: Z6 = AbelianGroupWithValues([-1], [6], names='g')
sage: g = Z6.gen(0)
sage: g.value()
-1
sage: g*g
```

(continues on next page)

(continued from previous page)

```
g^2
sage: (g*g).value()
1
sage: for i in range(7):
....:     print((i, g^i, (g^i).value()))
(0, 1, 1)
(1, g, -1)
(2, g^2, 1)
(3, g^3, -1)
(4, g^4, 1)
(5, g^5, -1)
(6, 1, 1)
```

```
>>> from sage.all import *
>>> Z6 = AbelianGroupWithValues([-Integer(1)], [Integer(6)], names='g')
>>> g = Z6.gen(Integer(0))
>>> g.value()
-1
>>> g*g
g^2
>>> (g*g).value()
1
>>> for i in range(Integer(7)):
...     print((i, g**i, (g**i).value()))
(0, 1, 1)
(1, g, -1)
(2, g^2, 1)
(3, g^3, -1)
(4, g^4, 1)
(5, g^5, -1)
(6, 1, 1)
```

The elements come with a coercion embedding into the `values_group()`, so you can use the group elements instead of the values:

```
sage: # needs sage.rings.number_field
sage: CF3.<zeta> = CyclotomicField(3)
sage: Z3.<g> = AbelianGroupWithValues([zeta], [3])
sage: Z3.values_group()
Cyclotomic Field of order 3 and degree 2
sage: g.value()
zeta
sage: CF3(g)
zeta
sage: g + zeta
2*zeta
sage: zeta + g
2*zeta
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> CF3 = CyclotomicField(Integer(3), names=('zeta',)); (zeta,) = CF3._first_ngens(1)
```

(continues on next page)

(continued from previous page)

```
>>> Z3 = AbelianGroupWithValues([zeta], [Integer(3)], names=('g',)); (g,) = Z3._first_
↪ngens(1)
>>> Z3.values_group()
Cyclotomic Field of order 3 and degree 2
>>> g.value()
zeta
>>> CF3(g)
zeta
>>> g + zeta
2*zeta
>>> zeta + g
2*zeta
```

sage.groups.abelian_gps.values.**AbelianGroupWithValues**(*values*, *n*, *gens_orders=None*, *names='f'*, *check=False*, *values_group=None*)

Construct an Abelian group with values associated to the generators.

INPUT:

- `values` – list/tuple/iterable of values that you want to associate to the generators
- `n` – integer (optional); if not specified, will be derived from `gens_orders`
- **gens_orders – list of nonnegative integers in the form**
  $[a_0, a_1, \dots, a_{n-1}]$, typically written in increasing order. This list is padded with zeros if it has length less than n. The orders of the commuting generators, with 0 denoting an infinite cyclic factor.
- `names` – (optional) names of generators
- `values_group` – a parent or `None` (default). The common parent of the values. This might be a group, but can also just contain the values. For example, if the values are units in a ring then the `values_group` would be the whole ring. If `None` it will be derived from the values.

EXAMPLES:

```
sage: G = AbelianGroupWithValues([-1], [6])
sage: g = G.gen(0)
sage: for i in range(7):
....:     print((i, g^i, (g^i).value()))
(0, 1, 1)
(1, f, -1)
(2, f^2, 1)
(3, f^3, -1)
(4, f^4, 1)
(5, f^5, -1)
(6, 1, 1)
sage: G.values_group()
Integer Ring
```

```
>>> from sage.all import *
>>> G = AbelianGroupWithValues([-Integer(1)], [Integer(6)])
>>> g = G.gen(Integer(0))
>>> for i in range(Integer(7)):
...     print((i, g**i, (g**i).value()))
(0, 1, 1)
```

(continues on next page)

(continued from previous page)

```
(1, f, -1)
(2, f^2, 1)
(3, f^3, -1)
(4, f^4, 1)
(5, f^5, -1)
(6, 1, 1)
>>> G.values_group()
Integer Ring
```

The group elements come with a coercion embedding into the `values_group()`, so you can use them like their `value()`

```
sage: G.values_embedding()
Generic morphism:
  From: Multiplicative Abelian group isomorphic to C6
  To:   Integer Ring
sage: g.value()
-1
sage: 0 + g
-1
sage: 1 + 2*g
-1
```

```
>>> from sage.all import *
>>> G.values_embedding()
Generic morphism:
  From: Multiplicative Abelian group isomorphic to C6
  To:   Integer Ring
>>> g.value()
-1
>>> Integer(0) + g
-1
>>> Integer(1) + Integer(2)*g
-1
```

class sage.groups.abelian_gps.values.**AbelianGroupWithValuesElement**(*parent*, *exponents*, *value=None*)

Bases: *AbelianGroupElement*

An element of an Abelian group with values assigned to generators.

INPUT:

- `exponents` – tuple of integers; the exponent vector defining the group element
- `parent` – the parent
- `value` – the value assigned to the group element or `None` (default); in the latter case, the value is computed as needed

EXAMPLES:

```
sage: F = AbelianGroupWithValues([1,-1], [2,4])
sage: a,b = F.gens()
sage: TestSuite(a*b).run()
```

```
>>> from sage.all import *
>>> F = AbelianGroupWithValues([Integer(1),-Integer(1)], [Integer(2),Integer(4)])
>>> a,b = F.gens()
>>> TestSuite(a*b).run()
```

value()

Return the value of the group element.

OUTPUT: the value according to the values for generators; see `gens_values()`

EXAMPLES:

```
sage: G = AbelianGroupWithValues([5], 1)
sage: G.0.value()
5
```

```
>>> from sage.all import *
>>> G = AbelianGroupWithValues([Integer(5)], Integer(1))
>>> G.gen(0).value()
5
```

class sage.groups.abelian_gps.values.**AbelianGroupWithValuesEmbedding**(*domain*, *codomain*)

Bases: `Morphism`

The morphism embedding the Abelian group with values in its values group.

INPUT:

- `domain` – a *AbelianGroupWithValues_class*
- `codomain` – the values group (need not be in the category of groups, e.g. symbolic ring)

EXAMPLES:

```
sage: # needs sage.symbolic
sage: Z4.<g> = AbelianGroupWithValues([I], [4])
sage: embedding = Z4.values_embedding(); embedding
Generic morphism:
  From: Multiplicative Abelian group isomorphic to C4
  To:   Number Field in I with defining polynomial x^2 + 1 with I = 1*I
sage: embedding(1)
1
sage: embedding(g)
I
sage: embedding(g^2)
-1
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> Z4 = AbelianGroupWithValues([I], [Integer(4)], names=('g',)); (g,) = Z4._
↪first_ngens(1)
>>> embedding = Z4.values_embedding(); embedding
Generic morphism:
  From: Multiplicative Abelian group isomorphic to C4
  To:   Number Field in I with defining polynomial x^2 + 1 with I = 1*I
```

(continues on next page)

(continued from previous page)

```
>>> embedding(Integer(1))
1
>>> embedding(g)
I
>>> embedding(g**Integer(2))
-1
```

class sage.groups.abelian_gps.values.**AbelianGroupWithValues_class**(*generator_orders*, *names*, *values*, *values_group*)

Bases: *AbelianGroup_class*

The class of an Abelian group with values associated to the generator.

INPUT:

- `generator_orders` – tuple of integers; the orders of the generators
- `names` – string or list of strings; the names for the generators
- `values` – tuple the same length as the number of generators; the values assigned to the generators
- `values_group` – the common parent of the values

EXAMPLES:

```
sage: G.<a,b> = AbelianGroupWithValues([2,-1], [0,4])
sage: TestSuite(G).run()
```

```
>>> from sage.all import *
>>> G = AbelianGroupWithValues([Integer(2),-Integer(1)], [Integer(0),Integer(4)],
↪names=('a', 'b',)); (a, b,) = G._first_ngens(2)
>>> TestSuite(G).run()
```

Element

alias of *AbelianGroupWithValuesElement*

gen(*i=0*)

The i-th generator of the abelian group.

INPUT:

- `i` – integer (default: 0); the index of the generator

OUTPUT: a group element

EXAMPLES:

```
sage: F = AbelianGroupWithValues([1,2,3,4,5], 5, [], names='a')
sage: F.0
a0
sage: F.0.value()
1
sage: F.2
a2
sage: F.2.value()
3
```

(continues on next page)

(continued from previous page)

```
sage: G = AbelianGroupWithValues([-1,0,1], [2,1,3])
sage: G.gens()
(f0, 1, f2)
```

```
>>> from sage.all import *
>>> F = AbelianGroupWithValues([Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)], Integer(5), [], names='a')
>>> F.gen(0)
a0
>>> F.gen(0).value()
1
>>> F.gen(2)
a2
>>> F.gen(2).value()
3

>>> G = AbelianGroupWithValues([-Integer(1),Integer(0),Integer(1)],␣
↪[Integer(2),Integer(1),Integer(3)])
>>> G.gens()
(f0, 1, f2)
```

gens_values()

Return the values associated to the generators.

OUTPUT: tuple

EXAMPLES:

```
sage: G = AbelianGroupWithValues([-1,0,1], [2,1,3])
sage: G.gens()
(f0, 1, f2)
sage: G.gens_values()
(-1, 0, 1)
```

```
>>> from sage.all import *
>>> G = AbelianGroupWithValues([-Integer(1),Integer(0),Integer(1)],␣
↪[Integer(2),Integer(1),Integer(3)])
>>> G.gens()
(f0, 1, f2)
>>> G.gens_values()
(-1, 0, 1)
```

values_embedding()

Return the embedding of `self` in *values_group()*.

OUTPUT: a morphism

EXAMPLES:

```
sage: Z4 = AbelianGroupWithValues([I], [4])                                 #␣
↪needs sage.symbolic
sage: Z4.values_embedding()                                                 #␣
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
Generic morphism:
  From: Multiplicative Abelian group isomorphic to C4
  To:   Number Field in I with defining polynomial x^2 + 1 with I = 1*I
```

```
>>> from sage.all import *
>>> Z4 = AbelianGroupWithValues([I], [Integer(4)])                     ␣
↪     # needs sage.symbolic
>>> Z4.values_embedding()                                              #␣
↪needs sage.symbolic
Generic morphism:
  From: Multiplicative Abelian group isomorphic to C4
  To:   Number Field in I with defining polynomial x^2 + 1 with I = 1*I
```

values_group()

The common parent of the values.

The values need to form a multiplicative group, but can be embedded in a larger structure. For example, if the values are units in a ring then the `values_group()` would be the whole ring.

OUTPUT: the common parent of the values, containing the group generated by all values

EXAMPLES:

```
sage: G = AbelianGroupWithValues([-1,0,1], [2,1,3])
sage: G.values_group()
Integer Ring

sage: Z4 = AbelianGroupWithValues([I], [4])                              #␣
↪needs sage.symbolic
sage: Z4.values_group()                                                  #␣
↪needs sage.symbolic
Number Field in I with defining polynomial x^2 + 1 with I = 1*I
```

```
>>> from sage.all import *
>>> G = AbelianGroupWithValues([-Integer(1),Integer(0),Integer(1)],␣
↪[Integer(2),Integer(1),Integer(3)])
>>> G.values_group()
Integer Ring

>>> Z4 = AbelianGroupWithValues([I], [Integer(4)])                     ␣
↪     # needs sage.symbolic
>>> Z4.values_group()                                                  #␣
↪needs sage.symbolic
Number Field in I with defining polynomial x^2 + 1 with I = 1*I
```

26.5 Dual groups of Finite Multiplicative Abelian Groups

The basic idea is very simple. Let G be an abelian group and G^* its dual (i.e., the group of homomorphisms from G to $\mathbf{C}^\times$). Let g_j, $j = 1, .., n$, denote generators of G - say g_j is of order $m_j > 1$. There are generators X_j, $j = 1, .., n$, of G^* for which $X_j(g_j) = \exp(2\pi i/m_j)$ and $X_i(g_j) = 1$ if $i \neq j$. These are used to construct G^*.

Sage supports multiplicative abelian groups on any prescribed finite number $n > 0$ of generators. Use `AbelianGroup()` function to create an abelian group, the `dual_group()` method to create its dual, and then the `gen()` and `gens()`

methods to obtain the corresponding generators. You can print the generators as arbitrary strings using the optional `names` argument to the *dual_group()* method.

EXAMPLES:

```
sage: F = AbelianGroup(5, [2,5,7,8,9], names='abcde')
sage: (a, b, c, d, e) = F.gens()

sage: Fd = F.dual_group(names='ABCDE')
sage: Fd.base_ring()
Cyclotomic Field of order 2520 and degree 576
sage: A,B,C,D,E = Fd.gens()
sage: A(a)
-1
sage: A(b), A(c), A(d), A(e)
(1, 1, 1, 1)

sage: # needs sage.rings.real_mpfr
sage: Fd = F.dual_group(names='ABCDE', base_ring=CC)
sage: Fd.category()
Category of commutative groups
sage: A,B,C,D,E = Fd.gens()
sage: A(a)    # abs tol 1e-8
-1.00000000000000 + 0.00000000000000*I
sage: A(b); A(c); A(d); A(e)
1.00000000000000
1.00000000000000
1.00000000000000
1.00000000000000
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(5), [Integer(2),Integer(5),Integer(7),Integer(8),
↪Integer(9)], names='abcde')
>>> (a, b, c, d, e) = F.gens()

>>> Fd = F.dual_group(names='ABCDE')
>>> Fd.base_ring()
Cyclotomic Field of order 2520 and degree 576
>>> A,B,C,D,E = Fd.gens()
>>> A(a)
-1
>>> A(b), A(c), A(d), A(e)
(1, 1, 1, 1)

>>> # needs sage.rings.real_mpfr
>>> Fd = F.dual_group(names='ABCDE', base_ring=CC)
>>> Fd.category()
Category of commutative groups
>>> A,B,C,D,E = Fd.gens()
>>> A(a)    # abs tol 1e-8
-1.00000000000000 + 0.00000000000000*I
>>> A(b); A(c); A(d); A(e)
1.00000000000000
1.00000000000000
```

(continues on next page)

(continued from previous page)

```
1.00000000000000
1.00000000000000
```

AUTHORS:

- David Joyner (2006-08) (based on abelian_groups)
- David Joyner (2006-10) modifications suggested by William Stein
- Volker Braun (2012-11) port to new Parent base. Use tuples for immutables. Default to cyclotomic base ring.

class sage.groups.abelian_gps.dual_abelian_group.**DualAbelianGroup_class**(*G*, *names*, *base_ring*)

Bases: UniqueRepresentation, *AbelianGroup*

Dual of abelian group.

EXAMPLES:

```
sage: F = AbelianGroup(5,[3,5,7,8,9], names='abcde')
sage: F.dual_group()
Dual of Abelian Group isomorphic to Z/3Z x Z/5Z x Z/7Z x Z/8Z x Z/9Z
over Cyclotomic Field of order 2520 and degree 576

sage: F = AbelianGroup(4,[15,7,8,9], names='abcd')
sage: F.dual_group(base_ring=CC)                                         #␣
↪needs sage.rings.real_mpfr
Dual of Abelian Group isomorphic to Z/15Z x Z/7Z x Z/8Z x Z/9Z
over Complex Field with 53 bits of precision
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(5),[Integer(3),Integer(5),Integer(7),Integer(8),
↪Integer(9)], names='abcde')
>>> F.dual_group()
Dual of Abelian Group isomorphic to Z/3Z x Z/5Z x Z/7Z x Z/8Z x Z/9Z
over Cyclotomic Field of order 2520 and degree 576

>>> F = AbelianGroup(Integer(4),[Integer(15),Integer(7),Integer(8),Integer(9)],␣
↪names='abcd')
>>> F.dual_group(base_ring=CC)                                           #␣
↪needs sage.rings.real_mpfr
Dual of Abelian Group isomorphic to Z/15Z x Z/7Z x Z/8Z x Z/9Z
over Complex Field with 53 bits of precision
```

Element

alias of *DualAbelianGroupElement*

base_ring()

Return the scalars over which the group is dualized.

EXAMPLES:

```
sage: F = AbelianGroup(3,[5,64,729], names=list("abc"))
sage: Fd = F.dual_group(base_ring=CC)
sage: Fd.base_ring()
Complex Field with 53 bits of precision
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(3),[Integer(5),Integer(64),Integer(729)],␣
↪names=list("abc"))
>>> Fd = F.dual_group(base_ring=CC)
>>> Fd.base_ring()
Complex Field with 53 bits of precision
```

gen(*i=0*)

The i-th generator of the abelian group.

EXAMPLES:

```
sage: F = AbelianGroup(3, [1,2,3], names='a')
sage: Fd = F.dual_group(names='A')
sage: Fd.0
1
sage: Fd.1
A1
sage: Fd.gens_orders()
(1, 2, 3)
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(3), [Integer(1),Integer(2),Integer(3)], names='a
↪')
>>> Fd = F.dual_group(names='A')
>>> Fd.gen(0)
1
>>> Fd.gen(1)
A1
>>> Fd.gens_orders()
(1, 2, 3)
```

gens()

Return the generators for the group.

OUTPUT: tuple of group elements generating the group

EXAMPLES:

```
sage: F = AbelianGroup([7,11]).dual_group()
sage: F.gens()
(X0, X1)
```

```
>>> from sage.all import *
>>> F = AbelianGroup([Integer(7),Integer(11)]).dual_group()
>>> F.gens()
(X0, X1)
```

gens_orders()

The orders of the generators of the dual group.

OUTPUT: tuple of integers

EXAMPLES:

```
sage: F = AbelianGroup([5]*1000)
sage: Fd = F.dual_group()
sage: invs = Fd.gens_orders(); len(invs)
1000
```

```
>>> from sage.all import *
>>> F = AbelianGroup([Integer(5)]*Integer(1000))
>>> Fd = F.dual_group()
>>> invs = Fd.gens_orders(); len(invs)
1000
```

group()

Return the group that `self` is the dual of.

EXAMPLES:

```
sage: F = AbelianGroup(3,[5,64,729], names=list("abc"))
sage: Fd = F.dual_group(base_ring=CC)
sage: Fd.group() is F
True
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(3),[Integer(5),Integer(64),Integer(729)],␣
↪names=list("abc"))
>>> Fd = F.dual_group(base_ring=CC)
>>> Fd.group() is F
True
```

invariants()

The invariants of the dual group.

You should use *gens_orders()* instead.

EXAMPLES:

```
sage: F = AbelianGroup([5]*1000)
sage: Fd = F.dual_group()
sage: invs = Fd.gens_orders(); len(invs)
1000
```

```
>>> from sage.all import *
>>> F = AbelianGroup([Integer(5)]*Integer(1000))
>>> Fd = F.dual_group()
>>> invs = Fd.gens_orders(); len(invs)
1000
```

is_commutative()

Return `True` since this group is commutative.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,9])
sage: Gd = G.dual_group()
sage: Gd.is_commutative()
```

(continues on next page)

(continued from previous page)

```
True
sage: Gd.is_abelian()
True
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(2),Integer(3),Integer(9)])
>>> Gd = G.dual_group()
>>> Gd.is_commutative()
True
>>> Gd.is_abelian()
True
```

list()

Return a tuple of all elements of this group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3], names='ab')
sage: Gd = G.dual_group(names='AB')
sage: Gd.list()
(1, B, B^2, A, A*B, A*B^2)
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(2),Integer(3)], names='ab')
>>> Gd = G.dual_group(names='AB')
>>> Gd.list()
(1, B, B^2, A, A*B, A*B^2)
```

ngens()

The number of generators of the dual group.

EXAMPLES:

```
sage: F = AbelianGroup([7]*100)
sage: Fd = F.dual_group()
sage: Fd.ngens()
100
```

```
>>> from sage.all import *
>>> F = AbelianGroup([Integer(7)]*Integer(100))
>>> Fd = F.dual_group()
>>> Fd.ngens()
100
```

order()

Return the order of this group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,9])
sage: Gd = G.dual_group()
sage: Gd.order()
54
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(2),Integer(3),Integer(9)])
>>> Gd = G.dual_group()
>>> Gd.order()
54
```

random_element()

Return a random element of this dual group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,9])
sage: Gd = G.dual_group(base_ring=CC)                                  #␣
↪needs sage.rings.real_mpfr
sage: Gd.random_element().parent() is Gd                               #␣
↪needs sage.rings.real_mpfr
True

sage: # needs sage.rings.real_mpfr
sage: N = 43^2 - 1
sage: G = AbelianGroup([N], names='a')
sage: Gd = G.dual_group(names='A', base_ring=CC)
sage: a, = G.gens()
sage: A, = Gd.gens()
sage: x = a^(N/4); y = a^(N/3); z = a^(N/14)
sage: found = [False]*4
sage: while not all(found):
....:     X = A*Gd.random_element()
....:     found[len([b for b in [x,y,z] if abs(X(b)-1)>10^(-8)])] = True
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(2),Integer(3),Integer(9)])
>>> Gd = G.dual_group(base_ring=CC)                                   #␣
↪needs sage.rings.real_mpfr
>>> Gd.random_element().parent() is Gd                                #␣
↪needs sage.rings.real_mpfr
True

>>> # needs sage.rings.real_mpfr
>>> N = Integer(43)**Integer(2) - Integer(1)
>>> G = AbelianGroup([N], names='a')
>>> Gd = G.dual_group(names='A', base_ring=CC)
>>> a, = G.gens()
>>> A, = Gd.gens()
>>> x = a**(N/Integer(4)); y = a**(N/Integer(3)); z = a**(N/Integer(14))
>>> found = [False]*Integer(4)
>>> while not all(found):
...     X = A*Gd.random_element()
...     found[len([b for b in [x,y,z] if abs(X(b)-Integer(1))>Integer(10)**(-
↪Integer(8))])] = True
```

sage.groups.abelian_gps.dual_abelian_group.**is_DualAbelianGroup**(x)

Return `True` if x is the dual group of an abelian group.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.dual_abelian_group import is_DualAbelianGroup
sage: F = AbelianGroup(5,[3,5,7,8,9], names=list("abcde"))
sage: Fd = F.dual_group()
sage: is_DualAbelianGroup(Fd)
doctest:warning...
DeprecationWarning: the function is_DualAbelianGroup is deprecated;
use 'isinstance(..., DualAbelianGroup_class)' instead
See https://github.com/sagemath/sage/issues/37898 for details.
True
sage: F = AbelianGroup(3,[1,2,3], names='a')
sage: Fd = F.dual_group()
sage: Fd.gens()
(1, X1, X2)
sage: F.gens()
(1, a1, a2)
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.dual_abelian_group import is_DualAbelianGroup
>>> F = AbelianGroup(Integer(5),[Integer(3),Integer(5),Integer(7),Integer(8),
↪Integer(9)], names=list("abcde"))
>>> Fd = F.dual_group()
>>> is_DualAbelianGroup(Fd)
doctest:warning...
DeprecationWarning: the function is_DualAbelianGroup is deprecated;
use 'isinstance(..., DualAbelianGroup_class)' instead
See https://github.com/sagemath/sage/issues/37898 for details.
True
>>> F = AbelianGroup(Integer(3),[Integer(1),Integer(2),Integer(3)], names='a')
>>> Fd = F.dual_group()
>>> Fd.gens()
(1, X1, X2)
>>> F.gens()
(1, a1, a2)
```

26.6 Base class for abelian group elements

This is the base class for both `abelian_group_element` and `dual_abelian_group_element`.

As always, elements are immutable once constructed.

class `sage.groups.abelian_gps.element_base.`**`AbelianGroupElementBase`**(*parent*, *exponents*)

Bases: `MultiplicativeGroupElement`

Base class for abelian group elements.

The group element is defined by a tuple whose `i`-th entry is an integer in the range from 0 (inclusively) to `G.gen(i).order()` (exclusively) if the i-th generator is of finite order, and an arbitrary integer if the i-th generator is of infinite order.

INPUT:

- `exponents` – `1` or a list/tuple/iterable of integers; the exponent vector (with respect to the parent generators) defining the group element

- `parent` – abelian group; the parent of the group element

EXAMPLES:

```
sage: F = AbelianGroup(3,[7,8,9])
sage: Fd = F.dual_group(names='ABC')                                        #␣
↪needs sage.rings.number_field
sage: A,B,C = Fd.gens()                                                     #␣
↪needs sage.rings.number_field
sage: A*B^-1 in Fd                                                          #␣
↪needs sage.rings.number_field
True
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(3),[Integer(7),Integer(8),Integer(9)])
>>> Fd = F.dual_group(names='ABC')                                        #␣
↪needs sage.rings.number_field
>>> A,B,C = Fd.gens()                                                     #␣
↪needs sage.rings.number_field
>>> A*B**-Integer(1) in Fd                                                ␣
↪      # needs sage.rings.number_field
True
```

exponents()

The exponents of the generators defining the group element.

OUTPUT:

A tuple of integers for an abelian group element. The integer can be arbitrary if the corresponding generator has infinite order. If the generator is of finite order, the integer is in the range from 0 (inclusive) to the order (exclusive).

EXAMPLES:

```
sage: F.<a,b,c,f> = AbelianGroup([7,8,9,0])
sage: (a^3*b^2*c).exponents()
(3, 2, 1, 0)
sage: F([3, 2, 1, 0])
a^3*b^2*c
sage: (c^42).exponents()
(0, 0, 6, 0)
sage: (f^42).exponents()
(0, 0, 0, 42)
```

```
>>> from sage.all import *
>>> F = AbelianGroup([Integer(7),Integer(8),Integer(9),Integer(0)], names=('a
↪', 'b', 'c', 'f',)); (a, b, c, f,) = F._first_ngens(4)
>>> (a**Integer(3)*b**Integer(2)*c).exponents()
(3, 2, 1, 0)
>>> F([Integer(3), Integer(2), Integer(1), Integer(0)])
a^3*b^2*c
>>> (c**Integer(42)).exponents()
(0, 0, 6, 0)
>>> (f**Integer(42)).exponents()
(0, 0, 0, 42)
```

is_trivial()

Test whether `self` is the trivial group element `1`.

OUTPUT: boolean

EXAMPLES:

```
sage: G.<a,b> = AbelianGroup([0,5])
sage: (a^5).is_trivial()
False
sage: (b^5).is_trivial()
True
```

```
>>> from sage.all import *
>>> G = AbelianGroup([Integer(0),Integer(5)], names=('a', 'b',)); (a, b,) = G.
↪_first_ngens(2)
>>> (a**Integer(5)).is_trivial()
False
>>> (b**Integer(5)).is_trivial()
True
```

list()

Return a copy of the exponent vector.

Use *exponents()* instead.

OUTPUT:

The underlying coordinates used to represent this element. If this is a word in an abelian group on n generators, then this is a list of nonnegative integers of length n.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: F = AbelianGroup(5,[2, 3, 5, 7, 8], names='abcde')
sage: a,b,c,d,e = F.gens()
sage: Ad = F.dual_group(names='ABCDE')
sage: A,B,C,D,E = Ad.gens()
sage: (A*B*C^2*D^20*E^65).exponents()
(1, 1, 2, 6, 1)
sage: X = A*B*C^2*D^2*E^-6
sage: X.exponents()
(1, 1, 2, 2, 2)
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> F = AbelianGroup(Integer(5),[Integer(2), Integer(3), Integer(5),␣
↪Integer(7), Integer(8)], names='abcde')
>>> a,b,c,d,e = F.gens()
>>> Ad = F.dual_group(names='ABCDE')
>>> A,B,C,D,E = Ad.gens()
>>> (A*B*C**Integer(2)*D**Integer(20)*E**Integer(65)).exponents()
(1, 1, 2, 6, 1)
>>> X = A*B*C**Integer(2)*D**Integer(2)*E**-Integer(6)
>>> X.exponents()
(1, 1, 2, 2, 2)
```

multiplicative_order()

Return the order of this element.

OUTPUT: integer or `infinity`

EXAMPLES:

```
sage: F = AbelianGroup(3,[7,8,9])
sage: Fd = F.dual_group()                                          #␣
↪needs sage.rings.number_field
sage: A,B,C = Fd.gens()                                            #␣
↪needs sage.rings.number_field
sage: (B*C).order()                                                #␣
↪needs sage.rings.number_field
72

sage: F = AbelianGroup(3,[7,8,9]); F
Multiplicative Abelian group isomorphic to C7 x C8 x C9
sage: F.gens()[2].order()
9
sage: a,b,c = F.gens()
sage: (b*c).order()
72
sage: G = AbelianGroup(3,[7,8,9])
sage: type((G.0 * G.1).order())==Integer
True
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(3),[Integer(7),Integer(8),Integer(9)])
>>> Fd = F.dual_group()                                            #␣
↪needs sage.rings.number_field
>>> A,B,C = Fd.gens()                                              #␣
↪needs sage.rings.number_field
>>> (B*C).order()                                                  #␣
↪needs sage.rings.number_field
72

>>> F = AbelianGroup(Integer(3),[Integer(7),Integer(8),Integer(9)]); F
Multiplicative Abelian group isomorphic to C7 x C8 x C9
>>> F.gens()[Integer(2)].order()
9
>>> a,b,c = F.gens()
>>> (b*c).order()
72
>>> G = AbelianGroup(Integer(3),[Integer(7),Integer(8),Integer(9)])
>>> type((G.gen(0) * G.gen(1)).order())==Integer
True
```

order()

Return the order of this element.

OUTPUT: integer or `infinity`

EXAMPLES:

```
sage: F = AbelianGroup(3,[7,8,9])
sage: Fd = F.dual_group()                                                      #␣
↪needs sage.rings.number_field
sage: A,B,C = Fd.gens()                                                        #␣
↪needs sage.rings.number_field
sage: (B*C).order()                                                            #␣
↪needs sage.rings.number_field
72

sage: F = AbelianGroup(3,[7,8,9]); F
Multiplicative Abelian group isomorphic to C7 x C8 x C9
sage: F.gens()[2].order()
9
sage: a,b,c = F.gens()
sage: (b*c).order()
72
sage: G = AbelianGroup(3,[7,8,9])
sage: type((G.0 * G.1).order())==Integer
True
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(3),[Integer(7),Integer(8),Integer(9)])
>>> Fd = F.dual_group()                                                      #␣
↪needs sage.rings.number_field
>>> A,B,C = Fd.gens()                                                        #␣
↪needs sage.rings.number_field
>>> (B*C).order()                                                            #␣
↪needs sage.rings.number_field
72

>>> F = AbelianGroup(Integer(3),[Integer(7),Integer(8),Integer(9)]); F
Multiplicative Abelian group isomorphic to C7 x C8 x C9
>>> F.gens()[Integer(2)].order()
9
>>> a,b,c = F.gens()
>>> (b*c).order()
72
>>> G = AbelianGroup(Integer(3),[Integer(7),Integer(8),Integer(9)])
>>> type((G.gen(0) * G.gen(1)).order())==Integer
True
```

26.7 Abelian group elements

AUTHORS:

- David Joyner (2006-02); based on free_abelian_monoid_element.py, written by David Kohel.
- David Joyner (2006-05); bug fix in order
- David Joyner (2006-08); bug fix+new method in pow for negatives+fixed corresponding examples.
- David Joyner (2009-02): Fixed bug in order.
- Volker Braun (2012-11) port to new Parent base. Use tuples for immutables.

EXAMPLES:

Recall an example from abelian groups:

```
sage: F = AbelianGroup(5,[4,5,5,7,8],names = list("abcde"))
sage: (a,b,c,d,e) = F.gens()
sage: x = a*b^2*e*d^20*e^12
sage: x
a*b^2*d^6*e^5
sage: x = a^10*b^12*c^13*d^20*e^12
sage: x
a^2*b^2*c^3*d^6*e^4
sage: y = a^13*b^19*c^23*d^27*e^72
sage: y
a*b^4*c^3*d^6
sage: x*y
a^3*b*c*d^5*e^4
sage: x.list()
[2, 2, 3, 6, 4]
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(5),[Integer(4),Integer(5),Integer(5),Integer(7),
↪Integer(8)],names = list("abcde"))
>>> (a,b,c,d,e) = F.gens()
>>> x = a*b**Integer(2)*e*d**Integer(20)*e**Integer(12)
>>> x
a*b^2*d^6*e^5
>>> x = a**Integer(10)*b**Integer(12)*c**Integer(13)*d**Integer(20)*e**Integer(12)
>>> x
a^2*b^2*c^3*d^6*e^4
>>> y = a**Integer(13)*b**Integer(19)*c**Integer(23)*d**Integer(27)*e**Integer(72)
>>> y
a*b^4*c^3*d^6
>>> x*y
a^3*b*c*d^5*e^4
>>> x.list()
[2, 2, 3, 6, 4]
```

class sage.groups.abelian_gps.abelian_group_element.**AbelianGroupElement**(*parent*, *exponents*)

Bases: *AbelianGroupElementBase*

Elements of an *AbelianGroup*

INPUT:

- x – list/tuple/iterable of integers (the element vector)
- parent – the parent *AbelianGroup*

EXAMPLES:

```
sage: F = AbelianGroup(5, [3,4,5,8,7], 'abcde')
sage: a, b, c, d, e = F.gens()
sage: a^2 * b^3 * a^2 * b^-4
a*b^3
sage: b^-11
```

(continues on next page)

(continued from previous page)

```
b
sage: a^-11
a
sage: a*b in F
True
```

```
>>> from sage.all import *
>>> F = AbelianGroup(Integer(5), [Integer(3),Integer(4),Integer(5),Integer(8),
↪Integer(7)], 'abcde')
>>> a, b, c, d, e = F.gens()
>>> a**Integer(2) * b**Integer(3) * a**Integer(2) * b**-Integer(4)
a*b^3
>>> b**-Integer(11)
b
>>> a**-Integer(11)
a
>>> a*b in F
True
```

as_permutation()

Return the element of the permutation group G (isomorphic to the abelian group A) associated to a in A.

EXAMPLES:

```
sage: G = AbelianGroup(3, [2,3,4], names='abc'); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
sage: a,b,c = G.gens()
sage: Gp = G.permutation_group(); Gp                                    #␣
↪needs sage.groups
Permutation Group with generators [(1,2), (3,4,5), (6,7,8,9)]
sage: a.as_permutation()                                                #␣
↪needs sage.libs.gap
(1,2)
sage: ap = a.as_permutation(); ap                                       #␣
↪needs sage.libs.gap
(1,2)
sage: ap in Gp                                                          #␣
↪needs sage.groups sage.libs.gap
True
```

```
>>> from sage.all import *
>>> G = AbelianGroup(Integer(3), [Integer(2),Integer(3),Integer(4)], names=
↪'abc'); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
>>> a,b,c = G.gens()
>>> Gp = G.permutation_group(); Gp                                    #␣
↪needs sage.groups
Permutation Group with generators [(1,2), (3,4,5), (6,7,8,9)]
>>> a.as_permutation()                                                #␣
↪needs sage.libs.gap
(1,2)
>>> ap = a.as_permutation(); ap                                       #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.gap
(1,2)
>>> ap in Gp                                                                 #␣
↪needs sage.groups sage.libs.gap
True
```

word_problem(*words*)

G and H are abelian groups, g in G, H is a subgroup of G generated by a list (words) of elements of G. If self is in H, return the expression for self as a word in the elements of (words).

This function does not solve the word problem in Sage. Rather it pushes it over to GAP, which has optimized (non-deterministic) algorithms for the word problem.

Warning

Don't use E (or other GAP-reserved letters) as a generator name.

Todo

This needs a rewrite - see stuff in the matrix_grp directory.

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: G = AbelianGroup(2, [2,3], names='xy')
sage: x,y = G.gens()
sage: x.word_problem([x,y])
[[x, 1]]
sage: y.word_problem([x,y])
[[y, 1]]
sage: v = (y*x).word_problem([x,y]); v  # random
[[x, 1], [y, 1]]
sage: prod([x^i for x,i in v]) == y*x
True
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> G = AbelianGroup(Integer(2), [Integer(2),Integer(3)], names='xy')
>>> x,y = G.gens()
>>> x.word_problem([x,y])
[[x, 1]]
>>> y.word_problem([x,y])
[[y, 1]]
>>> v = (y*x).word_problem([x,y]); v  # random
[[x, 1], [y, 1]]
>>> prod([x**i for x,i in v]) == y*x
True
```

sage.groups.abelian_gps.abelian_group_element.**is_AbelianGroupElement**(*x*)

Return True if x is an abelian group element, i.e., an element of type *AbelianGroupElement*.

EXAMPLES: Though the integer 3 is in the integers, and the integers have an abelian group structure, 3 is not an AbelianGroupElement:

```
sage: from sage.groups.abelian_gps.abelian_group_element import is_
↪AbelianGroupElement
sage: is_AbelianGroupElement(3)
doctest:warning...
DeprecationWarning: The function is_AbelianGroupElement is deprecated;
use 'isinstance(..., AbelianGroupElement)' instead.
See https://github.com/sagemath/sage/issues/38184 for details.
False
sage: F = AbelianGroup(5, [3,4,5,8,7], 'abcde')
sage: is_AbelianGroupElement(F.0)
True
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.abelian_group_element import is_
↪AbelianGroupElement
>>> is_AbelianGroupElement(Integer(3))
doctest:warning...
DeprecationWarning: The function is_AbelianGroupElement is deprecated;
use 'isinstance(..., AbelianGroupElement)' instead.
See https://github.com/sagemath/sage/issues/38184 for details.
False
>>> F = AbelianGroup(Integer(5), [Integer(3),Integer(4),Integer(5),Integer(8),
↪Integer(7)], 'abcde')
>>> is_AbelianGroupElement(F.gen(0))
True
```

26.8 Elements (characters) of the dual group of a finite Abelian group

To obtain the dual group of a finite Abelian group, use the `dual_group()` method:

```
sage: F = AbelianGroup([2,3,5,7,8], names='abcde')
sage: F
Multiplicative Abelian group isomorphic to C2 x C3 x C5 x C7 x C8

sage: Fd = F.dual_group(names='ABCDE'); Fd
Dual of Abelian Group isomorphic to Z/2Z x Z/3Z x Z/5Z x Z/7Z x Z/8Z
over Cyclotomic Field of order 840 and degree 192
```

```
>>> from sage.all import *
>>> F = AbelianGroup([Integer(2),Integer(3),Integer(5),Integer(7),Integer(8)], names=
↪'abcde')
>>> F
Multiplicative Abelian group isomorphic to C2 x C3 x C5 x C7 x C8

>>> Fd = F.dual_group(names='ABCDE'); Fd
Dual of Abelian Group isomorphic to Z/2Z x Z/3Z x Z/5Z x Z/7Z x Z/8Z
over Cyclotomic Field of order 840 and degree 192
```

The elements of the dual group can be evaluated on elements of the original group:

```
sage: a,b,c,d,e = F.gens()
sage: A,B,C,D,E = Fd.gens()
sage: A*B^2*D^7
A*B^2
sage: A(a)
-1
sage: B(b)
zeta840^140 - 1
sage: CC(_)     # abs tol 1e-8
-0.499999999999995 + 0.866025403784447*I
sage: A(a*b)
-1
sage: (A*B*C^2*D^20*E^65).exponents()
(1, 1, 2, 6, 1)
sage: B^(-1)
B^2
```

```
>>> from sage.all import *
>>> a,b,c,d,e = F.gens()
>>> A,B,C,D,E = Fd.gens()
>>> A*B**Integer(2)*D**Integer(7)
A*B^2
>>> A(a)
-1
>>> B(b)
zeta840^140 - 1
>>> CC(_)     # abs tol 1e-8
-0.499999999999995 + 0.866025403784447*I
>>> A(a*b)
-1
>>> (A*B*C**Integer(2)*D**Integer(20)*E**Integer(65)).exponents()
(1, 1, 2, 6, 1)
>>> B**(-Integer(1))
B^2
```

AUTHORS:

- David Joyner (2006-07); based on abelian_group_element.py.
- David Joyner (2006-10); modifications suggested by William Stein.
- Volker Braun (2012-11) port to new Parent base. Use tuples for immutables. Default to cyclotomic base ring.

class sage.groups.abelian_gps.dual_abelian_group_element.**DualAbelianGroupElement**(*parent*, *exponents*)

Bases: *AbelianGroupElementBase*

Base class for abelian group elements

word_problem(*words*)

This is a rather hackish method and is included for completeness.

The word problem for an instance of DualAbelianGroup as it can for an AbelianGroup. The reason why is that word problem for an instance of AbelianGroup simply calls GAP (which has abelian groups implemented) and invokes "EpimorphismFromFreeGroup" and "PreImagesRepresentative". GAP does not have duals of abelian groups implemented. So, by using the same name for the generators, the method below converts the problem for the dual group to the corresponding problem on the group itself and uses GAP to solve that.

EXAMPLES:

```
sage: G = AbelianGroup(5,[3, 5, 5, 7, 8], names='abcde')
sage: Gd = G.dual_group(names='abcde')
sage: a,b,c,d,e = Gd.gens()
sage: u = a^3*b*c*d^2*e^5
sage: v = a^2*b*c^2*d^3*e^3
sage: w = a^7*b^3*c^5*d^4*e^4
sage: x = a^3*b^2*c^2*d^3*e^5
sage: y = a^2*b^4*c^2*d^4*e^5
sage: e.word_problem([u,v,w,x,y])                                   #␣
↪needs sage.libs.gap
[[b^2*c^2*d^3*e^5, 245]]
```

```
>>> from sage.all import *
>>> G = AbelianGroup(Integer(5),[Integer(3), Integer(5), Integer(5),␣
↪Integer(7), Integer(8)], names='abcde')
>>> Gd = G.dual_group(names='abcde')
>>> a,b,c,d,e = Gd.gens()
>>> u = a**Integer(3)*b*c*d**Integer(2)*e**Integer(5)
>>> v = a**Integer(2)*b*c**Integer(2)*d**Integer(3)*e**Integer(3)
>>> w = a**Integer(7)*b**Integer(3)*c**Integer(5)*d**Integer(4)*e**Integer(4)
>>> x = a**Integer(3)*b**Integer(2)*c**Integer(2)*d**Integer(3)*e**Integer(5)
>>> y = a**Integer(2)*b**Integer(4)*c**Integer(2)*d**Integer(4)*e**Integer(5)
>>> e.word_problem([u,v,w,x,y])                                     #␣
↪needs sage.libs.gap
[[b^2*c^2*d^3*e^5, 245]]
```

`sage.groups.abelian_gps.dual_abelian_group_element.`**`is_DualAbelianGroupElement`**`(x)`

Test whether `x` is a dual Abelian group element.

INPUT:

- `x` – anything

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.groups.abelian_gps.dual_abelian_group import is_
↪DualAbelianGroupElement
sage: F = AbelianGroup(5, [5,5,7,8,9], names=list("abcde")).dual_group()
sage: is_DualAbelianGroupElement(F)
doctest:warning...
DeprecationWarning: The function is_DualAbelianGroupElement is deprecated;
use 'isinstance(..., DualAbelianGroupElement)' instead.
See https://github.com/sagemath/sage/issues/38184 for details.
False
sage: is_DualAbelianGroupElement(F.an_element())
True
```

```
>>> from sage.all import *
>>> from sage.groups.abelian_gps.dual_abelian_group import is_
↪DualAbelianGroupElement
>>> F = AbelianGroup(Integer(5), [Integer(5),Integer(5),Integer(7),Integer(8),
↪Integer(9)], names=list("abcde")).dual_group()
>>> is_DualAbelianGroupElement(F)
doctest:warning...
DeprecationWarning: The function is_DualAbelianGroupElement is deprecated;
use 'isinstance(..., DualAbelianGroupElement)' instead.
See https://github.com/sagemath/sage/issues/38184 for details.
False
>>> is_DualAbelianGroupElement(F.an_element())
True
```

26.9 Homomorphisms of abelian groups

Todo

- there must be a homspace first
- there should be hom and Hom methods in abelian group

AUTHORS:

- David Joyner (2006-03-03): initial version

class sage.groups.abelian_gps.abelian_group_morphism.**AbelianGroupMap**(*parent*)

Bases: Morphism

A set-theoretic map between AbelianGroups.

class sage.groups.abelian_gps.abelian_group_morphism.**AbelianGroupMorphism**(*G*, *H*, *genss*, *imgss*)

Bases: Morphism

Some python code for wrapping GAP's `GroupHomomorphismByImages` function for abelian groups. Returns "fail" if `gens` does not generate `self` or if the map does not extend to a group homomorphism, `self` - `other`.

EXAMPLES:

```
sage: G = AbelianGroup(3,[2,3,4],names='abc'); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
sage: a,b,c = G.gens()
sage: H = AbelianGroup(2,[2,3],names='xy'); H
Multiplicative Abelian group isomorphic to C2 x C3
sage: x,y = H.gens()

sage: from sage.groups.abelian_gps.abelian_group_morphism import␣
↪AbelianGroupMorphism
sage: phi = AbelianGroupMorphism(H,G,[x,y],[a,b])  # optional - gap_package_
↪polycyclic
```

```
>>> from sage.all import *
>>> G = AbelianGroup(Integer(3),[Integer(2),Integer(3),Integer(4)],names='abc'); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
>>> a,b,c = G.gens()
>>> H = AbelianGroup(Integer(2),[Integer(2),Integer(3)],names='xy'); H
Multiplicative Abelian group isomorphic to C2 x C3
>>> x,y = H.gens()

>>> from sage.groups.abelian_gps.abelian_group_morphism import␣
↪AbelianGroupMorphism
>>> phi = AbelianGroupMorphism(H,G,[x,y],[a,b])  # optional - gap_package_
↪polycyclic
```

AUTHORS:

- David Joyner (2006-02)

image(*S*)

Return the image of the subgroup `S` by the morphism.

This only works for finite groups.

INPUT:

- `S` – a subgroup of the domain group `G`

EXAMPLES:

```
sage: G = AbelianGroup(2,[2,3],names='xy')
sage: x,y = G.gens()
sage: subG = G.subgroup([x])                          # optional - gap_package_
↪polycyclic
sage: H = AbelianGroup(3,[2,3,4],names='abc')
sage: a,b,c = H.gens()
sage: phi = AbelianGroupMorphism(G,H,[x,y],[a,b])   # optional - gap_package_
↪polycyclic
sage: phi.image(subG)                                 # optional - gap_package_
↪polycyclic
Multiplicative Abelian subgroup isomorphic to C2 generated by {a}
```

```
>>> from sage.all import *
>>> G = AbelianGroup(Integer(2),[Integer(2),Integer(3)],names='xy')
>>> x,y = G.gens()
>>> subG = G.subgroup([x])                          # optional - gap_package_
↪polycyclic
>>> H = AbelianGroup(Integer(3),[Integer(2),Integer(3),Integer(4)],names='abc
↪')
>>> a,b,c = H.gens()
>>> phi = AbelianGroupMorphism(G,H,[x,y],[a,b])   # optional - gap_package_
↪polycyclic
>>> phi.image(subG)                                 # optional - gap_package_
↪polycyclic
Multiplicative Abelian subgroup isomorphic to C2 generated by {a}
```

kernel()

Only works for finite groups.

Todo

not done yet; returns a gap object but should return a Sage group.

EXAMPLES:

```
sage: H = AbelianGroup(3,[2,3,4],names='abc'); H
Multiplicative Abelian group isomorphic to C2 x C3 x C4
sage: a,b,c = H.gens()
sage: G = AbelianGroup(2,[2,3],names='xy'); G
Multiplicative Abelian group isomorphic to C2 x C3
sage: x,y = G.gens()
sage: phi = AbelianGroupMorphism(G,H,[x,y],[a,b])   # optional - gap_package_
↪polycyclic
sage: phi.kernel()                                  # optional - gap_package_
↪polycyclic
Group([  ])

sage: H = AbelianGroup(3,[2,2,2],names='abc')
sage: a,b,c = H.gens()
sage: G = AbelianGroup(2,[2,2],names='x')
sage: x,y = G.gens()
sage: phi = AbelianGroupMorphism(G,H,[x,y],[a,a])   # optional - gap_package_
↪polycyclic
sage: phi.kernel()                                  # optional - gap_package_
↪polycyclic
Group([ f1*f2 ])
```

```
>>> from sage.all import *
>>> H = AbelianGroup(Integer(3),[Integer(2),Integer(3),Integer(4)],names='abc
↪'); H
Multiplicative Abelian group isomorphic to C2 x C3 x C4
>>> a,b,c = H.gens()
>>> G = AbelianGroup(Integer(2),[Integer(2),Integer(3)],names='xy'); G
Multiplicative Abelian group isomorphic to C2 x C3
>>> x,y = G.gens()
>>> phi = AbelianGroupMorphism(G,H,[x,y],[a,b])  # optional - gap_package_
↪polycyclic
>>> phi.kernel()                                 # optional - gap_package_
↪polycyclic
Group([  ])

>>> H = AbelianGroup(Integer(3),[Integer(2),Integer(2),Integer(2)],names='abc
↪')
>>> a,b,c = H.gens()
>>> G = AbelianGroup(Integer(2),[Integer(2),Integer(2)],names='x')
>>> x,y = G.gens()
>>> phi = AbelianGroupMorphism(G,H,[x,y],[a,a])  # optional - gap_package_
↪polycyclic
>>> phi.kernel()                                 # optional - gap_package_
↪polycyclic
Group([ f1*f2 ])
```

`sage.groups.abelian_gps.abelian_group_morphism.`**`is_AbelianGroupMorphism`**(f)

26.10 Additive Abelian Groups

Additive abelian groups are just modules over $\mathbf{Z}$. Hence the classes in this module derive from those in the module `sage.modules.fg_pid`. The only major differences are in the way elements are printed.

`sage.groups.additive_abelian.additive_abelian_group.`**`AdditiveAbelianGroup`**(*invs*, *remember_generators=True*)

Construct a finitely-generated additive abelian group.

INPUT:

- `invs` – list of integers; the invariants. These should all be greater than or equal to zero.
- `remember_generators` – boolean (default: `True`); whether or not to fix a set of generators (corresponding to the given invariants, which need not be in Smith form)

OUTPUT: the abelian group $\bigoplus_i \mathbf{Z}/n_i\mathbf{Z}$, where n_i are the invariants

EXAMPLES:

```
sage: AdditiveAbelianGroup([0, 2, 4])
Additive abelian group isomorphic to Z + Z/2 + Z/4
```

```
>>> from sage.all import *
>>> AdditiveAbelianGroup([Integer(0), Integer(2), Integer(4)])
Additive abelian group isomorphic to Z + Z/2 + Z/4
```

An example of the `remember_generators` switch:

```
sage: G = AdditiveAbelianGroup([0, 2, 3]); G
Additive abelian group isomorphic to Z + Z/2 + Z/3
sage: G.gens()
((1, 0, 0), (0, 1, 0), (0, 0, 1))

sage: H = AdditiveAbelianGroup([0, 2, 3], remember_generators = False); H
Additive abelian group isomorphic to Z/6 + Z
sage: H.gens()
((0, 1, 1), (1, 0, 0))
```

```
>>> from sage.all import *
>>> G = AdditiveAbelianGroup([Integer(0), Integer(2), Integer(3)]); G
Additive abelian group isomorphic to Z + Z/2 + Z/3
>>> G.gens()
((1, 0, 0), (0, 1, 0), (0, 0, 1))

>>> H = AdditiveAbelianGroup([Integer(0), Integer(2), Integer(3)], remember_
↪generators = False); H
Additive abelian group isomorphic to Z/6 + Z
>>> H.gens()
((0, 1, 1), (1, 0, 0))
```

There are several ways to create elements of an additive abelian group. Realize that there are two sets of generators: the "obvious" ones composed of zeros and ones, one for each invariant given to construct the group, the other being a set of minimal generators. Which set is the default varies with the use of the `remember_generators` switch.

First with "obvious" generators. Note that a raw list will use the minimal generators and a vector (a module element) will use the generators that pair up naturally with the invariants. We create the same element repeatedly.

```
sage: H = AdditiveAbelianGroup([3,2,0], remember_generators=True)
sage: H.gens()
((1, 0, 0), (0, 1, 0), (0, 0, 1))
sage: [H.0, H.1, H.2]
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
sage: p = H.0+H.1+6*H.2; p
(1, 1, 6)

sage: H.smith_form_gens()
((2, 1, 0), (0, 0, 1))
sage: q = H.linear_combination_of_smith_form_gens([5,6]); q
(1, 1, 6)
sage: p == q
True

sage: r = H(vector([1,1,6])); r
(1, 1, 6)
sage: p == r
True

sage: s = H(p)
sage: p == s
True
```

```
>>> from sage.all import *
>>> H = AdditiveAbelianGroup([Integer(3),Integer(2),Integer(0)], remember_
↪generators=True)
>>> H.gens()
((1, 0, 0), (0, 1, 0), (0, 0, 1))
>>> [H.gen(0), H.gen(1), H.gen(2)]
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
>>> p = H.gen(0)+H.gen(1)+Integer(6)*H.gen(2); p
(1, 1, 6)

>>> H.smith_form_gens()
((2, 1, 0), (0, 0, 1))
>>> q = H.linear_combination_of_smith_form_gens([Integer(5),Integer(6)]); q
(1, 1, 6)
>>> p == q
True

>>> r = H(vector([Integer(1),Integer(1),Integer(6)])); r
(1, 1, 6)
>>> p == r
True

>>> s = H(p)
```

(continues on next page)

(continued from previous page)

```
>>> p == s
True
```

Again, but now where the generators are the minimal set. Coercing a list or a vector works as before, but the default generators are different.

```
sage: G = AdditiveAbelianGroup([3,2,0], remember_generators=False)
sage: G.gens()
((2, 1, 0), (0, 0, 1))
sage: [G.0, G.1]
[(2, 1, 0), (0, 0, 1)]
sage: p = 5*G.0+6*G.1; p
(1, 1, 6)

sage: H.smith_form_gens()
((2, 1, 0), (0, 0, 1))
sage: q = G.linear_combination_of_smith_form_gens([5,6]); q
(1, 1, 6)
sage: p == q
True

sage: r = G(vector([1,1,6])); r
(1, 1, 6)
sage: p == r
True

sage: s = H(p)
sage: p == s
True
```

```
>>> from sage.all import *
>>> G = AdditiveAbelianGroup([Integer(3),Integer(2),Integer(0)], remember_
↪generators=False)
>>> G.gens()
((2, 1, 0), (0, 0, 1))
>>> [G.gen(0), G.gen(1)]
[(2, 1, 0), (0, 0, 1)]
>>> p = Integer(5)*G.gen(0)+Integer(6)*G.gen(1); p
(1, 1, 6)

>>> H.smith_form_gens()
((2, 1, 0), (0, 0, 1))
>>> q = G.linear_combination_of_smith_form_gens([Integer(5),Integer(6)]); q
(1, 1, 6)
>>> p == q
True

>>> r = G(vector([Integer(1),Integer(1),Integer(6)])); r
(1, 1, 6)
>>> p == r
True
```

(continues on next page)

(continued from previous page)

```
>>> s = H(p)
>>> p == s
True
```

class sage.groups.additive_abelian.additive_abelian_group.**AdditiveAbelianGroupElement**(*parent*, *x*, *check=True*)

Bases: `FGP_Element`

An element of an *AdditiveAbelianGroup_class*.

class sage.groups.additive_abelian.additive_abelian_group.**AdditiveAbelianGroup_class**(*cover*, *relations*)

Bases: `FGP_Module_class`

An additive abelian group, implemented using the $\mathbf{Z}$-module machinery.

INPUT:

- `cover` – the covering group as $\mathbf{Z}$-module
- `relations` – the relations as submodule of `cover`

Element

alias of *AdditiveAbelianGroupElement*

exponent()

Return the exponent of this group (the smallest positive integer N such that $Nx = 0$ for all x in the group). If there is no such integer, return 0.

EXAMPLES:

```
sage: AdditiveAbelianGroup([2,4]).exponent()
4
sage: AdditiveAbelianGroup([0, 2,4]).exponent()
0
sage: AdditiveAbelianGroup([]).exponent()
1
```

```
>>> from sage.all import *
>>> AdditiveAbelianGroup([Integer(2),Integer(4)]).exponent()
4
>>> AdditiveAbelianGroup([Integer(0), Integer(2),Integer(4)]).exponent()
0
>>> AdditiveAbelianGroup([]).exponent()
1
```

is_cyclic()

Return `True` if the group is cyclic.

EXAMPLES:

With no common factors between the orders of the generators, the group will be cyclic.

```
sage: G = AdditiveAbelianGroup([6, 7, 55])
sage: G.is_cyclic()
True
```

```
>>> from sage.all import *
>>> G = AdditiveAbelianGroup([Integer(6), Integer(7), Integer(55)])
>>> G.is_cyclic()
True
```

Repeating primes in the orders will create a non-cyclic group.

```
sage: G = AdditiveAbelianGroup([6, 15, 21, 33])
sage: G.is_cyclic()
False
```

```
>>> from sage.all import *
>>> G = AdditiveAbelianGroup([Integer(6), Integer(15), Integer(21),␣
↪Integer(33)])
>>> G.is_cyclic()
False
```

A trivial group is trivially cyclic.

```
sage: T = AdditiveAbelianGroup([1])
sage: T.is_cyclic()
True
```

```
>>> from sage.all import *
>>> T = AdditiveAbelianGroup([Integer(1)])
>>> T.is_cyclic()
True
```

is_multiplicative()

Return `False` since this is an additive group.

EXAMPLES:

```
sage: AdditiveAbelianGroup([0]).is_multiplicative()
False
```

```
>>> from sage.all import *
>>> AdditiveAbelianGroup([Integer(0)]).is_multiplicative()
False
```

order()

Return the order of this group (integer or infinity).

EXAMPLES:

```
sage: AdditiveAbelianGroup([2,4]).order()
8
sage: AdditiveAbelianGroup([0, 2,4]).order()
+Infinity
```

(continues on next page)

(continued from previous page)

```
sage: AdditiveAbelianGroup([]).order()
1
```

```
>>> from sage.all import *
>>> AdditiveAbelianGroup([Integer(2),Integer(4)]).order()
8
>>> AdditiveAbelianGroup([Integer(0), Integer(2),Integer(4)]).order()
+Infinity
>>> AdditiveAbelianGroup([]).order()
1
```

short_name()

Return a name for the isomorphism class of this group.

EXAMPLES:

```
sage: AdditiveAbelianGroup([0, 2,4]).short_name()
'Z + Z/2 + Z/4'
sage: AdditiveAbelianGroup([0, 2, 3]).short_name()
'Z + Z/2 + Z/3'
```

```
>>> from sage.all import *
>>> AdditiveAbelianGroup([Integer(0), Integer(2),Integer(4)]).short_name()
'Z + Z/2 + Z/4'
>>> AdditiveAbelianGroup([Integer(0), Integer(2), Integer(3)]).short_name()
'Z + Z/2 + Z/3'
```

class sage.groups.additive_abelian.additive_abelian_group.**AdditiveAbelianGroup_fixed_gens**(*cover*, *rels*, *gens*)

Bases: *AdditiveAbelianGroup_class*

A variant which fixes a set of generators, which need not be in Smith form (or indeed independent).

gens()

Return the specified generators for `self` (as a tuple). Compare `self.smithform_gens()`.

EXAMPLES:

```
sage: G = AdditiveAbelianGroup([2,3])
sage: G.gens()
((1, 0), (0, 1))
sage: G.smith_form_gens()
((1, 2),)
```

```
>>> from sage.all import *
>>> G = AdditiveAbelianGroup([Integer(2),Integer(3)])
>>> G.gens()
((1, 0), (0, 1))
>>> G.smith_form_gens()
((1, 2),)
```

identity()

Return the identity (zero) element of this group.

EXAMPLES:

```
sage: G = AdditiveAbelianGroup([2, 3])
sage: G.identity()
(0, 0)
```

```
>>> from sage.all import *
>>> G = AdditiveAbelianGroup([Integer(2), Integer(3)])
>>> G.identity()
(0, 0)
```

permutation_group()

Return the permutation group attached to this group.

EXAMPLES:

```
sage: G = AdditiveAbelianGroup([2, 3])
sage: G.permutation_group()                                                 #␣
↪needs sage.groups
Permutation Group with generators [(1,2), (3,4,5)]
```

```
>>> from sage.all import *
>>> G = AdditiveAbelianGroup([Integer(2), Integer(3)])
>>> G.permutation_group()                                                 #␣
↪needs sage.groups
Permutation Group with generators [(1,2), (3,4,5)]
```

sage.groups.additive_abelian.additive_abelian_group.**cover_and_relations_from_invariants**(*invs*)

A utility function to construct modules required to initialize the super class.

Given a list of integers, this routine constructs the obvious pair of free modules such that the quotient of the two free modules over $\mathbf{Z}$ is naturally isomorphic to the corresponding product of cyclic modules (and hence isomorphic to a direct sum of cyclic groups).

EXAMPLES:

```
sage: from sage.groups.additive_abelian.additive_abelian_group import cover_and_
↪relations_from_invariants as cr
sage: cr([0,2,3])
(Ambient free module of rank 3 over the principal ideal domain Integer Ring, Free␣
↪module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[0 2 0]
[0 0 3])
```

```
>>> from sage.all import *
>>> from sage.groups.additive_abelian.additive_abelian_group import cover_and_
↪relations_from_invariants as cr
>>> cr([Integer(0),Integer(2),Integer(3)])
(Ambient free module of rank 3 over the principal ideal domain Integer Ring, Free␣
↪module of degree 3 and rank 2 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
Echelon basis matrix:
[0 2 0]
[0 0 3])
```

26.11 Wrapper class for abelian groups

This class is intended as a template for anything in Sage that needs the functionality of abelian groups. One can create an `AdditiveAbelianGroupWrapper` object from any given set of elements in some given parent, as long as an `_add_` method has been defined.

EXAMPLES:

We create a toy example based on the Mordell-Weil group of an elliptic curve over **Q**:

```
sage: # needs sage.schemes
sage: E = EllipticCurve('30a2')
sage: pts = [E(4,-7,1), E(7/4, -11/8, 1), E(3, -2, 1)]
sage: M = AdditiveAbelianGroupWrapper(pts[0].parent(), pts, [3, 2, 2]); M
Additive abelian group isomorphic to Z/3 + Z/2 + Z/2 embedded in Abelian
group of points on Elliptic Curve defined by y^2 + x*y + y = x^3 - 19*x + 26
over Rational Field
sage: M.gens()
((4 : -7 : 1), (7/4 : -11/8 : 1), (3 : -2 : 1))
sage: 3*M.0
(0 : 1 : 0)
sage: 3000000000000001 * M.0
(4 : -7 : 1)
sage: M == loads(dumps(M))  # known bug (https://github.com/sagemath/sage/issues/11599
↪#comment:7)
True
```

```
>>> from sage.all import *
>>> # needs sage.schemes
>>> E = EllipticCurve('30a2')
>>> pts = [E(Integer(4),-Integer(7),Integer(1)), E(Integer(7)/Integer(4), -
↪Integer(11)/Integer(8), Integer(1)), E(Integer(3), -Integer(2), Integer(1))]
>>> M = AdditiveAbelianGroupWrapper(pts[Integer(0)].parent(), pts, [Integer(3),␣
↪Integer(2), Integer(2)]); M
Additive abelian group isomorphic to Z/3 + Z/2 + Z/2 embedded in Abelian
group of points on Elliptic Curve defined by y^2 + x*y + y = x^3 - 19*x + 26
over Rational Field
>>> M.gens()
((4 : -7 : 1), (7/4 : -11/8 : 1), (3 : -2 : 1))
>>> Integer(3)*M.gen(0)
(0 : 1 : 0)
>>> Integer(3000000000000001) * M.gen(0)
(4 : -7 : 1)
>>> M == loads(dumps(M))  # known bug (https://github.com/sagemath/sage/issues/11599
↪#comment:7)
True
```

Todo

Think about subgroups and quotients, which probably won't work in the current implementation – some fiddly adjustments will be needed in order to be able to pass extra arguments to the subquotient's init method.

AUTHORS:

- David Loeffler (2010)
- Lorenz Panny (2017): *AdditiveAbelianGroupWrapper.discrete_log()*
- Lorenz Panny (2023): *AdditiveAbelianGroupWrapper.from_generators()*

class sage.groups.additive_abelian.additive_abelian_wrapper.**AdditiveAbelianGroupWrapper**(*universe*, *gens*, *invariants*)

Bases: *AdditiveAbelianGroup_fixed_gens*

This class is used to wrap a subgroup of an existing additive abelian group as a new additive abelian group.

EXAMPLES:

```
sage: G2 = AdditiveAbelianGroupWrapper(Zmod(42), [2], [21]); G2
Additive abelian group isomorphic to Z/21 embedded in Ring of integers modulo 42
sage: G6 = AdditiveAbelianGroupWrapper(Zmod(42), [6], [7]); G6
Additive abelian group isomorphic to Z/7 embedded in Ring of integers modulo 42
sage: G = AdditiveAbelianGroupWrapper(Zmod(42), [21,14,6], [2,3,7]); G
Additive abelian group isomorphic to Z/2 + Z/3 + Z/7 embedded in
 Ring of integers modulo 42
sage: G.invariants()
(42,)
```

```
>>> from sage.all import *
>>> G2 = AdditiveAbelianGroupWrapper(Zmod(Integer(42)), [Integer(2)],
↪[Integer(21)]); G2
Additive abelian group isomorphic to Z/21 embedded in Ring of integers modulo 42
>>> G6 = AdditiveAbelianGroupWrapper(Zmod(Integer(42)), [Integer(6)],
↪[Integer(7)]); G6
Additive abelian group isomorphic to Z/7 embedded in Ring of integers modulo 42
>>> G = AdditiveAbelianGroupWrapper(Zmod(Integer(42)), [Integer(21),Integer(14),
↪Integer(6)], [Integer(2),Integer(3),Integer(7)]); G
Additive abelian group isomorphic to Z/2 + Z/3 + Z/7 embedded in
 Ring of integers modulo 42
>>> G.invariants()
(42,)
```

```
sage: AdditiveAbelianGroupWrapper(QQbar, [sqrt(2), sqrt(3)], [0, 0])          #
↪needs sage.rings.number_field sage.symbolic
Additive abelian group isomorphic to Z + Z embedded in Algebraic Field
```

```
>>> from sage.all import *
>>> AdditiveAbelianGroupWrapper(QQbar, [sqrt(Integer(2)), sqrt(Integer(3))],␣
↪[Integer(0), Integer(0)])           # needs sage.rings.number_field sage.
↪symbolic
Additive abelian group isomorphic to Z + Z embedded in Algebraic Field
```

```
sage: EllipticCurve(GF(419**2), [1,0]).abelian_group()  # indirect doctest      #␣
↪needs sage.rings.finite_rings sage.schemes
Additive abelian group isomorphic to Z/420 + Z/420 embedded in
 Abelian group of points on Elliptic Curve
  defined by y^2 = x^3 + x over Finite Field in z2 of size 419^2
```

```
>>> from sage.all import *
>>> EllipticCurve(GF(Integer(419)**Integer(2)), [Integer(1),Integer(0)]).abelian_
↪group()  # indirect doctest      # needs sage.rings.finite_rings sage.schemes
Additive abelian group isomorphic to Z/420 + Z/420 embedded in
 Abelian group of points on Elliptic Curve
  defined by y^2 = x^3 + x over Finite Field in z2 of size 419^2
```

Element

alias of *AdditiveAbelianGroupWrapperElement*

discrete_exp(*v*)

Given a list (or other iterable) of length equal to the number of generators of this group, compute the element of the ambient group with those exponents in terms of the generators of `self`.

EXAMPLES:

```
sage: G = AdditiveAbelianGroupWrapper(QQbar,                                   #␣
↪needs sage.rings.number_field
....:                                [sqrt(QQbar(2)), -1], [0, 0])
sage: v = G.discrete_exp([3, 5]); v                                            #␣
↪needs sage.rings.number_field
-0.7573593128807148?
sage: v.parent() is QQbar                                                      #␣
↪needs sage.rings.number_field
True
```

```
>>> from sage.all import *
>>> G = AdditiveAbelianGroupWrapper(QQbar,                                   #␣
↪needs sage.rings.number_field
...                                [sqrt(QQbar(Integer(2))), -Integer(1)],␣
↪[Integer(0), Integer(0)])
>>> v = G.discrete_exp([Integer(3), Integer(5)]); v                             ␣
↪              # needs sage.rings.number_field
-0.7573593128807148?
>>> v.parent() is QQbar                                                      #␣
↪needs sage.rings.number_field
True
```

This method is an inverse of *discrete_log()*:

```
sage: orders = [2, 2*3, 2*3*5, 2*3*5*7, 2*3*5*7*11]
sage: G = AdditiveAbelianGroup(orders)
sage: A = AdditiveAbelianGroupWrapper(G.0.parent(), G.gens(), orders)
sage: el = A.random_element()
sage: A.discrete_exp(A.discrete_log(el)) == el
True
```

```
>>> from sage.all import *
>>> orders = [Integer(2), Integer(2)*Integer(3),␣
↪Integer(2)*Integer(3)*Integer(5),␣
↪Integer(2)*Integer(3)*Integer(5)*Integer(7),␣
↪Integer(2)*Integer(3)*Integer(5)*Integer(7)*Integer(11)]
>>> G = AdditiveAbelianGroup(orders)
>>> A = AdditiveAbelianGroupWrapper(G.gen(0).parent(), G.gens(), orders)
>>> el = A.random_element()
>>> A.discrete_exp(A.discrete_log(el)) == el
True
```

discrete_log(*x*, *gens=None*)

Given an element of the ambient group, attempt to express it in terms of the generators of this group or the given generators of a subgroup.

ALGORITHM:

This reduces to p-groups, then calls `_discrete_log_pgroup()` which implements a basic version of the recursive algorithm from [Suth2008].

AUTHORS:

- Lorenz Panny (2017)

EXAMPLES:

```
sage: G = AdditiveAbelianGroup([2, 2*3, 2*3*5, 2*3*5*7, 2*3*5*7*11])
sage: A = AdditiveAbelianGroupWrapper(G.0.parent(), G.gens(),
....:                                 [g.order() for g in G.gens()])
sage: A.discrete_log(A.discrete_exp([1,5,23,127,539]))
(1, 5, 23, 127, 539)
```

```
>>> from sage.all import *
>>> G = AdditiveAbelianGroup([Integer(2), Integer(2)*Integer(3),␣
↪Integer(2)*Integer(3)*Integer(5),␣
↪Integer(2)*Integer(3)*Integer(5)*Integer(7),␣
↪Integer(2)*Integer(3)*Integer(5)*Integer(7)*Integer(11)])
>>> A = AdditiveAbelianGroupWrapper(G.gen(0).parent(), G.gens(),
...                                 [g.order() for g in G.gens()])
>>> A.discrete_log(A.discrete_exp([Integer(1),Integer(5),Integer(23),
↪Integer(127),Integer(539)]))
(1, 5, 23, 127, 539)
```

```
sage: x = polygen(ZZ, 'x')
sage: F.<t> = GF(1009**2, modulus=x**2+11); E = EllipticCurve(j=F(940))     #␣
↪needs sage.rings.finite_rings sage.schemes
sage: P, Q = E(900*t + 228, 974*t + 185), E(1007*t + 214, 865*t + 802)      #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.finite_rings sage.schemes
sage: E.abelian_group().discrete_log(123 * P + 777 * Q, [P, Q])          #␣
↪needs sage.rings.finite_rings sage.schemes
(123, 777)
```

```
>>> from sage.all import *
>>> x = polygen(ZZ, 'x')
>>> F = GF(Integer(1009)**Integer(2), modulus=x**Integer(2)+Integer(11),␣
↪names=('t',)); (t,) = F._first_ngens(1); E =␣
↪EllipticCurve(j=F(Integer(940)))      # needs sage.rings.finite_rings sage.
↪schemes
>>> P, Q = E(Integer(900)*t + Integer(228), Integer(974)*t + Integer(185)),␣
↪E(Integer(1007)*t + Integer(214), Integer(865)*t + Integer(802))      #␣
↪needs sage.rings.finite_rings sage.schemes
>>> E.abelian_group().discrete_log(Integer(123) * P + Integer(777) * Q, [P,␣
↪Q])             # needs sage.rings.finite_rings sage.schemes
(123, 777)
```

```
sage: V = Zmod(8)**2
sage: G = AdditiveAbelianGroupWrapper(V, [[2,2],[4,0]], [4, 2])
sage: G.discrete_log(V([6, 2]))
(1, 1)
sage: G.discrete_log(V([6, 4]))
Traceback (most recent call last):
...
ValueError: not in group
```

```
>>> from sage.all import *
>>> V = Zmod(Integer(8))**Integer(2)
>>> G = AdditiveAbelianGroupWrapper(V, [[Integer(2),Integer(2)],[Integer(4),
↪Integer(0)]], [Integer(4), Integer(2)])
>>> G.discrete_log(V([Integer(6), Integer(2)]))
(1, 1)
>>> G.discrete_log(V([Integer(6), Integer(4)]))
Traceback (most recent call last):
...
ValueError: not in group
```

```
sage: G = AdditiveAbelianGroupWrapper(QQbar, [sqrt(2)], [0])             #␣
↪needs sage.rings.number_field sage.symbolic
sage: G.discrete_log(QQbar(2*sqrt(2)))                                   #␣
↪needs sage.rings.number_field sage.symbolic
Traceback (most recent call last):
...
NotImplementedError: No black-box discrete log for infinite abelian groups
```

```
>>> from sage.all import *
>>> G = AdditiveAbelianGroupWrapper(QQbar, [sqrt(Integer(2))], [Integer(0)]) ␣
↪               # needs sage.rings.number_field sage.symbolic
>>> G.discrete_log(QQbar(Integer(2)*sqrt(Integer(2))))                      ␣
↪               # needs sage.rings.number_field sage.symbolic
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
NotImplementedError: No black-box discrete log for infinite abelian groups
```

static from_generators(*gens*, *universe=None*)

This method constructs the subgroup generated by a sequence of *finite-order* elements in an additive abelian group.

The elements need not be independent, hence this can be used to perform tasks such as finding relations between some given elements of an abelian group, computing the structure of the generated subgroup, enumerating all elements of the subgroup, and solving discrete-logarithm problems.

EXAMPLES:

```
sage: G = AdditiveAbelianGroup([15, 30, 45])
sage: gs = [G((1,2,3)), G((4,5,6)), G((7,7,7)), G((3,2,1))]
sage: H = AdditiveAbelianGroupWrapper.from_generators(gs); H
Additive abelian group isomorphic to Z/90 + Z/15 embedded in
 Additive abelian group isomorphic to Z/15 + Z/30 + Z/45
sage: H.gens()
((12, 13, 14), (1, 26, 21))
```

```
>>> from sage.all import *
>>> G = AdditiveAbelianGroup([Integer(15), Integer(30), Integer(45)])
>>> gs = [G((Integer(1),Integer(2),Integer(3))), G((Integer(4),Integer(5),
↪Integer(6))), G((Integer(7),Integer(7),Integer(7))), G((Integer(3),
↪Integer(2),Integer(1)))]
>>> H = AdditiveAbelianGroupWrapper.from_generators(gs); H
Additive abelian group isomorphic to Z/90 + Z/15 embedded in
 Additive abelian group isomorphic to Z/15 + Z/30 + Z/45
>>> H.gens()
((12, 13, 14), (1, 26, 21))
```

generator_orders()

The orders of the generators with which this group was initialised. (Note that these are not necessarily a minimal set of generators.) Generators of infinite order are returned as 0. Compare `self.invariants()`, which returns the orders of a minimal set of generators.

EXAMPLES:

```
sage: V = Zmod(6)**2
sage: G = AdditiveAbelianGroupWrapper(V, [2*V.0, 3*V.1], [3, 2])
sage: G.generator_orders()
(3, 2)
sage: G.invariants()
(6,)
```

```
>>> from sage.all import *
>>> V = Zmod(Integer(6))**Integer(2)
>>> G = AdditiveAbelianGroupWrapper(V, [Integer(2)*V.gen(0), Integer(3)*V.
↪gen(1)], [Integer(3), Integer(2)])
>>> G.generator_orders()
(3, 2)
```

(continues on next page)

(continued from previous page)

```
>>> G.invariants()
(6,)
```

torsion_subgroup(*n=None*)

Return the n-torsion subgroup of this additive abelian group when n is given, and the torsion subgroup otherwise.

The [n-]torsion subgroup consists of all elements whose order is finite [and divides n].

EXAMPLES:

```
sage: ords = [2, 2*3, 2*3*5, 0, 2*3*5*7, 2*3*5*7*11]
sage: G = AdditiveAbelianGroup(ords)
sage: A = AdditiveAbelianGroupWrapper(G.0.parent(), G.gens(), ords)
sage: T = A.torsion_subgroup(5)
sage: T
Additive abelian group isomorphic to Z/5 + Z/5 + Z/5 embedded in
 Additive abelian group isomorphic to Z/2 + Z/6 + Z/30 + Z + Z/210 + Z/2310
sage: T.gens()
((0, 0, 6, 0, 0, 0), (0, 0, 0, 0, 42, 0), (0, 0, 0, 0, 0, 462))
```

```
>>> from sage.all import *
>>> ords = [Integer(2), Integer(2)*Integer(3),␣
↪Integer(2)*Integer(3)*Integer(5), Integer(0),␣
↪Integer(2)*Integer(3)*Integer(5)*Integer(7),␣
↪Integer(2)*Integer(3)*Integer(5)*Integer(7)*Integer(11)]
>>> G = AdditiveAbelianGroup(ords)
>>> A = AdditiveAbelianGroupWrapper(G.gen(0).parent(), G.gens(), ords)
>>> T = A.torsion_subgroup(Integer(5))
>>> T
Additive abelian group isomorphic to Z/5 + Z/5 + Z/5 embedded in
 Additive abelian group isomorphic to Z/2 + Z/6 + Z/30 + Z + Z/210 + Z/2310
>>> T.gens()
((0, 0, 6, 0, 0, 0), (0, 0, 0, 0, 42, 0), (0, 0, 0, 0, 0, 462))
```

```
sage: # needs sage.rings.finite_rings sage.schemes
sage: E = EllipticCurve(GF(487^2), [311,205])
sage: T = E.abelian_group().torsion_subgroup(42); T
Additive abelian group isomorphic to Z/42 + Z/6 embedded in
 Abelian group of points on Elliptic Curve
  defined by y^2 = x^3 + 311*x + 205 over Finite Field in z2 of size 487^2
sage: [P.order() for P in T.gens()]
[42, 6]
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings sage.schemes
>>> E = EllipticCurve(GF(Integer(487)**Integer(2)), [Integer(311),
↪Integer(205)])
>>> T = E.abelian_group().torsion_subgroup(Integer(42)); T
Additive abelian group isomorphic to Z/42 + Z/6 embedded in
 Abelian group of points on Elliptic Curve
  defined by y^2 = x^3 + 311*x + 205 over Finite Field in z2 of size 487^2
```

(continues on next page)

(continued from previous page)

```
>>> [P.order() for P in T.gens()]
[42, 6]
```

```
sage: # needs sage.schemes
sage: E = EllipticCurve('574i1')
sage: pts = [E(103,172), E(61,18)]
sage: assert pts[0].order() == 7 and pts[1].order() == infinity
sage: M = AdditiveAbelianGroupWrapper(pts[0].parent(), pts, [7,0]); M
Additive abelian group isomorphic to Z/7 + Z embedded in
 Abelian group of points on Elliptic Curve defined by
  y^2 + x*y + y = x^3 - x^2 - 19353*x + 958713 over Rational Field
sage: M.torsion_subgroup()
Additive abelian group isomorphic to Z/7 embedded in
 Abelian group of points on Elliptic Curve defined by
  y^2 + x*y + y = x^3 - x^2 - 19353*x + 958713 over Rational Field
sage: M.torsion_subgroup(7)
Additive abelian group isomorphic to Z/7 embedded in
 Abelian group of points on Elliptic Curve defined by
  y^2 + x*y + y = x^3 - x^2 - 19353*x + 958713 over Rational Field
sage: M.torsion_subgroup(5)
Trivial group embedded in Abelian group of points on Elliptic Curve
 defined by y^2 + x*y + y = x^3 - x^2 - 19353*x + 958713 over Rational Field
```

```
>>> from sage.all import *
>>> # needs sage.schemes
>>> E = EllipticCurve('574i1')
>>> pts = [E(Integer(103),Integer(172)), E(Integer(61),Integer(18))]
>>> assert pts[Integer(0)].order() == Integer(7) and pts[Integer(1)].order()␣
↪== infinity
>>> M = AdditiveAbelianGroupWrapper(pts[Integer(0)].parent(), pts,␣
↪[Integer(7),Integer(0)]); M
Additive abelian group isomorphic to Z/7 + Z embedded in
 Abelian group of points on Elliptic Curve defined by
  y^2 + x*y + y = x^3 - x^2 - 19353*x + 958713 over Rational Field
>>> M.torsion_subgroup()
Additive abelian group isomorphic to Z/7 embedded in
 Abelian group of points on Elliptic Curve defined by
  y^2 + x*y + y = x^3 - x^2 - 19353*x + 958713 over Rational Field
>>> M.torsion_subgroup(Integer(7))
Additive abelian group isomorphic to Z/7 embedded in
 Abelian group of points on Elliptic Curve defined by
  y^2 + x*y + y = x^3 - x^2 - 19353*x + 958713 over Rational Field
>>> M.torsion_subgroup(Integer(5))
Trivial group embedded in Abelian group of points on Elliptic Curve
 defined by y^2 + x*y + y = x^3 - x^2 - 19353*x + 958713 over Rational Field
```

AUTHORS:

- Lorenz Panny (2022)

universe()

The ambient group in which this abelian group lives.

EXAMPLES:

```
sage: G = AdditiveAbelianGroupWrapper(QQbar,                                   #␣
↪needs sage.rings.number_field
....:                                 [sqrt(QQbar(2)), sqrt(QQbar(3))],
....:                                 [0, 0])
sage: G.universe()                                                             #␣
↪needs sage.rings.number_field
Algebraic Field
```

```
>>> from sage.all import *
>>> G = AdditiveAbelianGroupWrapper(QQbar,                                   #␣
↪needs sage.rings.number_field
...                                 [sqrt(QQbar(Integer(2))),␣
↪sqrt(QQbar(Integer(3)))],
...                                 [Integer(0), Integer(0)])
>>> G.universe()                                                             #␣
↪needs sage.rings.number_field
Algebraic Field
```

class sage.groups.additive_abelian.additive_abelian_wrapper.**AdditiveAbelianGroupWrapperElement**(*parent*, *vector*, *element=N*, *check=*

Bases: `AdditiveAbelianGroupElement`

An element of an `AdditiveAbelianGroupWrapper`.

element()

Return the underlying object that this element wraps.

EXAMPLES:

```
sage: T = EllipticCurve('65a').torsion_subgroup().gen(0)                       #␣
↪needs sage.schemes
sage: T; type(T)                                                               #␣
↪needs sage.schemes
(0 : 0 : 1)
<class 'sage.schemes.elliptic_curves.ell_torsion.EllipticCurveTorsionSubgroup_
↪with_category.element_class'>
sage: T.element(); type(T.element())                                           #␣
↪needs sage.schemes
(0 : 0 : 1)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field
↪'>
```

```
>>> from sage.all import *
>>> T = EllipticCurve('65a').torsion_subgroup().gen(Integer(0))              ␣
↪     # needs sage.schemes
>>> T; type(T)                                                               #␣
↪needs sage.schemes
```

(continues on next page)

(continued from previous page)

```
(0 : 0 : 1)
<class 'sage.schemes.elliptic_curves.ell_torsion.EllipticCurveTorsionSubgroup_
↪with_category.element_class'>
>>> T.element(); type(T.element())                                        #␣
↪needs sage.schemes
(0 : 0 : 1)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field
↪'>
```

class sage.groups.additive_abelian.additive_abelian_wrapper.**UnwrappingMorphism**(*domain*)

Bases: Morphism

The embedding into the ambient group. Used by the coercion framework.

sage.groups.additive_abelian.additive_abelian_wrapper.**basis_from_generators**(*gens*, *ords=None*)

Given a generating set of some finite abelian group (additively written), compute and return a basis of the group.

> **Note**
>
> A *basis* of a finite abelian group is a generating set $\{g_1, \ldots, g_n\}$ such that each element of the group can be written as a unique linear combination $\alpha_1 g_1 + \cdots + \alpha_n g_n$ with each $\alpha_i \in \{0, \ldots, \mathrm{ord}(g_i) - 1\}$.

ALGORITHM: [Suth2007], Algorithm 9.1 & Remark 9.1

EXAMPLES:

```
sage: # needs sage.groups sage.rings.finite_rings
sage: from sage.groups.additive_abelian.additive_abelian_wrapper import basis_
↪from_generators
sage: E = EllipticCurve(GF(31337^6,'a'), j=37)
sage: E.order()
946988065073788930380545280
sage: (R,S), (ordR,ordS) = basis_from_generators(E.gens())
sage: ordR, ordS
(313157428926517503432720, 3024)
sage: R.order() == ordR
True
sage: S.order() == ordS
True
sage: ordR * ordS == E.order()
True
sage: R.weil_pairing(S, ordR).multiplicative_order() == ordS
True
sage: E.abelian_group().invariants()
(3024, 313157428926517503432720)
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.finite_rings
>>> from sage.groups.additive_abelian.additive_abelian_wrapper import basis_from_
↪generators
>>> E = EllipticCurve(GF(Integer(31337)**Integer(6),'a'), j=Integer(37))
```

(continues on next page)

(continued from previous page)

```
>>> E.order()
946988065073788930380545280
>>> (R,S), (ordR,ordS) = basis_from_generators(E.gens())
>>> ordR, ordS
(313157428926517503432720, 3024)
>>> R.order() == ordR
True
>>> S.order() == ordS
True
>>> ordR * ordS == E.order()
True
>>> R.weil_pairing(S, ordR).multiplicative_order() == ordS
True
>>> E.abelian_group().invariants()
(3024, 313157428926517503432720)
```

26.12 Groups of elements representing (complex) arguments.

This includes

- *RootsOfUnityGroup* (containing all roots of unity)
- *UnitCircleGroup* (representing elements on the unit circle by $e^{2\pi \cdot \mathit{exponent}}$)
- *ArgumentByElementGroup* (whose elements are defined via formal arguments by $e^{I \cdot \arg(\mathit{element})}$.

Use the factory *ArgumentGroup* for creating such a group conveniently.

Note

One main purpose of such groups is in an `asymptotic ring's growth group` when an element like z^n (for some constant z) is split into $|z|^n \cdot e^{I \cdot \arg(z)n}$. (Note that the first factor determines the growth of that product, the second does not influence the growth.)

AUTHORS:

- Daniel Krenn (2018)

26.12.1 Classes and Methods

class sage.groups.misc_gps.argument_groups.**AbstractArgument**(*parent*, *element*, *normalize=True*)

Bases: `MultiplicativeGroupElement`

An element of *AbstractArgumentGroup*. This abstract class encapsulates an element of the parent's base, i.e. it can be seen as a wrapper class.

INPUT:

- `parent` – a SageMath parent
- `element` – an element of parent's base
- `normalize` – boolean (default: `True`)

class sage.groups.misc_gps.argument_groups.**AbstractArgumentGroup**(*base*, *category*)

Bases: UniqueRepresentation, Parent

A group whose elements represent (complex) arguments.

INPUT:

- base – a SageMath parent
- category – a category

Element

alias of *AbstractArgument*

class sage.groups.misc_gps.argument_groups.**ArgumentByElement**(*parent*, *element*, *normalize=True*)

Bases: *AbstractArgument*

An element of *ArgumentByElementGroup*.

INPUT:

- parent – a SageMath parent
- element – a nonzero element of the parent's base
- normalize – boolean (default: True)

class sage.groups.misc_gps.argument_groups.**ArgumentByElementGroup**(*base*, *category*)

Bases: *AbstractArgumentGroup*

A group of (complex) arguments. The arguments are represented by a the formal argument of an element, i.e., by arg(*element*).

INPUT:

- base – a SageMath parent representing a subset of the complex plane
- category – a category

EXAMPLES:

```
sage: from sage.groups.misc_gps.argument_groups import ArgumentByElementGroup
sage: C = ArgumentByElementGroup(CC); C
Unit Circle Group with Argument of Elements in
Complex Field with 53 bits of precision
sage: C(1 + 2*I)                                                        #␣
↪needs sage.symbolic
e^(I*arg(1.00000000000000 + 2.00000000000000*I))
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.argument_groups import ArgumentByElementGroup
>>> C = ArgumentByElementGroup(CC); C
Unit Circle Group with Argument of Elements in
Complex Field with 53 bits of precision
>>> C(Integer(1) + Integer(2)*I)                                        ␣
↪              # needs sage.symbolic
e^(I*arg(1.00000000000000 + 2.00000000000000*I))
```

Element

alias of *ArgumentByElement*

sage.groups.misc_gps.argument_groups.**ArgumentGroup = <sage.groups.misc_gps.argument_groups.ArgumentGroupFactory object>**

A factory for argument groups.

This is an instance of *ArgumentGroupFactory* whose documentation provides more details.

class sage.groups.misc_gps.argument_groups.**ArgumentGroupFactory**

Bases: UniqueFactory

A factory for creating argument groups.

INPUT:

- data – an object

  The factory will analyze data and interpret it as specification or domain.

- specification – string

  The following is possible:

  – 'Signs' give the *SignGroup*

  – 'UU' give the *RootsOfUnityGroup*

  – 'UU_P', where 'P' is a string representing a SageMath parent which is interpreted as exponents

  – 'Arg_P', where 'P' is a string representing a SageMath parent which is interpreted as domain

- domain – a SageMath parent representing a subset of the complex plane; an instance of *ArgumentByElementGroup* will be created with the given domain

- exponents – a SageMath parent representing a subset of the reals; an instance of :class`UnitCircleGroup` will be created with the given exponents

Exactly one of data, specification, exponents has to be provided.

Further keyword parameters will be carried on to the initialization of the group.

EXAMPLES:

```
sage: from sage.groups.misc_gps.argument_groups import ArgumentGroup

sage: ArgumentGroup('UU')                                               #␣
↪needs sage.rings.number_field
Group of Roots of Unity

sage: # needs sage.rings.number_field
sage: ArgumentGroup(ZZ)
Sign Group
sage: ArgumentGroup(QQ)
Sign Group
sage: ArgumentGroup('UU_QQ')
Group of Roots of Unity
sage: ArgumentGroup(AA)
Sign Group

sage: ArgumentGroup(RR)                                                 #␣
↪needs sage.rings.number_field
Sign Group
sage: ArgumentGroup('Arg_RR')                                           #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.number_field
Sign Group
sage: ArgumentGroup(RIF)                                                    #␣
↪needs sage.rings.real_interval_field
Sign Group
sage: ArgumentGroup(RBF)
Sign Group

sage: ArgumentGroup(CC)                                                     #␣
↪needs sage.rings.number_field
Unit Circle Group with Exponents in
Real Field with 53 bits of precision modulo ZZ
sage: ArgumentGroup('Arg_CC')                                               #␣
↪needs sage.rings.number_field
Unit Circle Group with Exponents in
Real Field with 53 bits of precision modulo ZZ
sage: ArgumentGroup(CIF)
Unit Circle Group with Exponents in
Real Interval Field with 53 bits of precision modulo ZZ
sage: ArgumentGroup(CBF)
Unit Circle Group with Exponents in
Real ball field with 53 bits of precision modulo ZZ

sage: ArgumentGroup(CyclotomicField(3))                                     #␣
↪needs sage.rings.number_field
Unit Circle Group with Argument of Elements in
Cyclotomic Field of order 3 and degree 2
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.argument_groups import ArgumentGroup

>>> ArgumentGroup('UU')                                                   #␣
↪needs sage.rings.number_field
Group of Roots of Unity

>>> # needs sage.rings.number_field
>>> ArgumentGroup(ZZ)
Sign Group
>>> ArgumentGroup(QQ)
Sign Group
>>> ArgumentGroup('UU_QQ')
Group of Roots of Unity
>>> ArgumentGroup(AA)
Sign Group

>>> ArgumentGroup(RR)                                                     #␣
↪needs sage.rings.number_field
Sign Group
>>> ArgumentGroup('Arg_RR')                                               #␣
↪needs sage.rings.number_field
Sign Group
>>> ArgumentGroup(RIF)                                                    #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.real_interval_field
Sign Group
>>> ArgumentGroup(RBF)
Sign Group

>>> ArgumentGroup(CC)                                                    #␣
↪needs sage.rings.number_field
Unit Circle Group with Exponents in
Real Field with 53 bits of precision modulo ZZ
>>> ArgumentGroup('Arg_CC')                                              #␣
↪needs sage.rings.number_field
Unit Circle Group with Exponents in
Real Field with 53 bits of precision modulo ZZ
>>> ArgumentGroup(CIF)
Unit Circle Group with Exponents in
Real Interval Field with 53 bits of precision modulo ZZ
>>> ArgumentGroup(CBF)
Unit Circle Group with Exponents in
Real ball field with 53 bits of precision modulo ZZ

>>> ArgumentGroup(CyclotomicField(Integer(3)))                             ␣
↪     # needs sage.rings.number_field
Unit Circle Group with Argument of Elements in
Cyclotomic Field of order 3 and degree 2
```

create_key_and_extra_args(*data=None*, *specification=None*, *domain=None*, *exponents=None*, ***kwds*)

Normalize the input.

See *ArgumentGroupFactory* for a description and examples.

create_object(*version*, *key*, ***kwds*)

Create an object from the given arguments.

class sage.groups.misc_gps.argument_groups.**RootOfUnity**(*parent*, *element*, *normalize=True*)

Bases: *UnitCirclePoint*

A root of unity (i.e. an element of *RootsOfUnityGroup*) which is $e^{2\pi \cdot exponent}$ for a rational `exponent`.

exponent_denominator()

Return the denominator of the rational quotient in $[0, 1)$ representing the exponent of this root of unity.

EXAMPLES:

```
sage: from sage.groups.misc_gps.argument_groups import RootsOfUnityGroup
sage: U = RootsOfUnityGroup()
sage: a = U(exponent=2/3); a
zeta3^2
sage: a.exponent_denominator()
3
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.argument_groups import RootsOfUnityGroup
>>> U = RootsOfUnityGroup()
>>> a = U(exponent=Integer(2)/Integer(3)); a
```

(continues on next page)

(continued from previous page)

```
zeta3^2
>>> a.exponent_denominator()
3
```

exponent_numerator()

Return the numerator of the rational quotient in $[0, 1)$ representing the exponent of this root of unity.

EXAMPLES:

```
sage: from sage.groups.misc_gps.argument_groups import RootsOfUnityGroup
sage: U = RootsOfUnityGroup()
sage: a = U(exponent=2/3); a
zeta3^2
sage: a.exponent_numerator()
2
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.argument_groups import RootsOfUnityGroup
>>> U = RootsOfUnityGroup()
>>> a = U(exponent=Integer(2)/Integer(3)); a
zeta3^2
>>> a.exponent_numerator()
2
```

class sage.groups.misc_gps.argument_groups.**RootsOfUnityGroup**(*category*)

Bases: *UnitCircleGroup*

The group of all roots of unity.

INPUT:

- category – a category

This is a specialized *UnitCircleGroup* with base **Q**.

EXAMPLES:

```
sage: from sage.groups.misc_gps.argument_groups import RootsOfUnityGroup
sage: U = RootsOfUnityGroup(); U
Group of Roots of Unity
sage: U(exponent=1/4)
I
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.argument_groups import RootsOfUnityGroup
>>> U = RootsOfUnityGroup(); U
Group of Roots of Unity
>>> U(exponent=Integer(1)/Integer(4))
I
```

Element

alias of *RootOfUnity*

class sage.groups.misc_gps.argument_groups.**Sign**(*parent*, *element*, *normalize=True*)

Bases: *AbstractArgument*

An element of *SignGroup*.

INPUT:

- `parent` – a SageMath parent
- `element` – a nonzero element of the parent's base
- `normalize` – boolean (default: `True`)

is_minus_one()

Return whether this sign is -1.

EXAMPLES:

```
sage: from sage.groups.misc_gps.argument_groups import SignGroup
sage: S = SignGroup()
sage: S(1).is_minus_one()
False
sage: S(-1).is_minus_one()
True
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.argument_groups import SignGroup
>>> S = SignGroup()
>>> S(Integer(1)).is_minus_one()
False
>>> S(-Integer(1)).is_minus_one()
True
```

is_one()

Return whether this sign is 1.

EXAMPLES:

```
sage: from sage.groups.misc_gps.argument_groups import SignGroup
sage: S = SignGroup()
sage: S(-1).is_one()
False
sage: S(1).is_one()
True
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.argument_groups import SignGroup
>>> S = SignGroup()
>>> S(-Integer(1)).is_one()
False
>>> S(Integer(1)).is_one()
True
```

class sage.groups.misc_gps.argument_groups.**SignGroup**(*category*)

Bases: *AbstractArgumentGroup*

A group of the signs -1 and 1.

INPUT:

- `category` – a category

EXAMPLES:

```
sage: from sage.groups.misc_gps.argument_groups import SignGroup
sage: S = SignGroup(); S
Sign Group
sage: S(-1)
-1
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.argument_groups import SignGroup
>>> S = SignGroup(); S
Sign Group
>>> S(-Integer(1))
-1
```

Element

alias of `Sign`

class `sage.groups.misc_gps.argument_groups.`**`UnitCircleGroup`**(*base*, *category*)

Bases: `AbstractArgumentGroup`

A group of points on the unit circle. These points are represented by $e^{2\pi \cdot exponent}$.

INPUT:

- `base` – a SageMath parent representing a subset of the reals
- `category` – a category

EXAMPLES:

```
sage: from sage.groups.misc_gps.argument_groups import UnitCircleGroup

sage: R = UnitCircleGroup(RR); R
Unit Circle Group with Exponents in Real Field with 53 bits of precision modulo ZZ
sage: R(exponent=2.42)
e^(2*pi*0.420000000000000)

sage: Q = UnitCircleGroup(QQ); Q
Unit Circle Group with Exponents in Rational Field modulo ZZ
sage: Q(exponent=6/5)
e^(2*pi*1/5)
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.argument_groups import UnitCircleGroup

>>> R = UnitCircleGroup(RR); R
Unit Circle Group with Exponents in Real Field with 53 bits of precision modulo ZZ
>>> R(exponent=RealNumber('2.42'))
e^(2*pi*0.420000000000000)

>>> Q = UnitCircleGroup(QQ); Q
Unit Circle Group with Exponents in Rational Field modulo ZZ
>>> Q(exponent=Integer(6)/Integer(5))
e^(2*pi*1/5)
```

Element

alias of `UnitCirclePoint`

class sage.groups.misc_gps.argument_groups.**UnitCirclePoint**(*parent*, *element*, *normalize=True*)

Bases: `AbstractArgument`

An element of `UnitCircleGroup` which is $e^{2\pi \cdot exponent}$.

INPUT:

- `parent` – a SageMath parent
- `exponent` – a number (of a subset of the reals)
- `normalize` – boolean (default: `True`)

property exponent

The exponent of this point on the unit circle.

EXAMPLES:

```
sage: from sage.groups.misc_gps.argument_groups import UnitCircleGroup
sage: C = UnitCircleGroup(RR)
sage: C(exponent=4/3).exponent
0.333333333333333
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.argument_groups import UnitCircleGroup
>>> C = UnitCircleGroup(RR)
>>> C(exponent=Integer(4)/Integer(3)).exponent
0.333333333333333
```

is_minus_one()

Return whether this point on the unit circle is -1.

EXAMPLES:

```
sage: from sage.groups.misc_gps.argument_groups import UnitCircleGroup
sage: C = UnitCircleGroup(QQ)
sage: C(exponent=0).is_minus_one()
False
sage: C(exponent=1/2).is_minus_one()
True
sage: C(exponent=2/3).is_minus_one()
False
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.argument_groups import UnitCircleGroup
>>> C = UnitCircleGroup(QQ)
>>> C(exponent=Integer(0)).is_minus_one()
False
>>> C(exponent=Integer(1)/Integer(2)).is_minus_one()
True
>>> C(exponent=Integer(2)/Integer(3)).is_minus_one()
False
```

is_one()

Return whether this point on the unit circle is 1.

EXAMPLES:

```
sage: from sage.groups.misc_gps.argument_groups import UnitCircleGroup
sage: C = UnitCircleGroup(QQ)
sage: C(exponent=0).is_one()
True
sage: C(exponent=1/2).is_one()
False
sage: C(exponent=2/3).is_one()
False
sage: C(exponent=42).is_one()
True
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.argument_groups import UnitCircleGroup
>>> C = UnitCircleGroup(QQ)
>>> C(exponent=Integer(0)).is_one()
True
>>> C(exponent=Integer(1)/Integer(2)).is_one()
False
>>> C(exponent=Integer(2)/Integer(3)).is_one()
False
>>> C(exponent=Integer(42)).is_one()
True
```

26.13 Groups of imaginary elements

Note

One main purpose of such groups is in an `asymptotic ring's growth group` when an element like n^z (for some constant z) is split into $n^{\Re z + I \Im z}$. (Note that the first summand in the exponent determines the growth, the second does not influence the growth.)

AUTHORS:

- Daniel Krenn (2018)

26.13.1 Classes and Methods

class sage.groups.misc_gps.imaginary_groups.**ImaginaryElement**(*parent*, *imag*)

Bases: `AdditiveGroupElement`

An element of *ImaginaryGroup*.

INPUT:

- `parent` – a SageMath parent
- `imag` – an element of parent's base

imag()

Return the imaginary part of this imaginary element.

EXAMPLES:

```
sage: from sage.groups.misc_gps.imaginary_groups import ImaginaryGroup
sage: J = ImaginaryGroup(ZZ)
sage: J(I).imag()                                                              #␣
↪needs sage.symbolic
1
sage: imag_part(J(I))   # indirect doctest                                     #␣
↪needs sage.symbolic
1
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.imaginary_groups import ImaginaryGroup
>>> J = ImaginaryGroup(ZZ)
>>> J(I).imag()                                                              #␣
↪needs sage.symbolic
1
>>> imag_part(J(I))   # indirect doctest                                     #␣
↪needs sage.symbolic
1
```

real()

Return the real part (= 0) of this imaginary element.

EXAMPLES:

```
sage: from sage.groups.misc_gps.imaginary_groups import ImaginaryGroup
sage: J = ImaginaryGroup(ZZ)
sage: J(I).real()                                                              #␣
↪needs sage.symbolic
0
sage: real_part(J(I))   # indirect doctest                                     #␣
↪needs sage.symbolic
0
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.imaginary_groups import ImaginaryGroup
>>> J = ImaginaryGroup(ZZ)
>>> J(I).real()                                                              #␣
↪needs sage.symbolic
0
>>> real_part(J(I))   # indirect doctest                                     #␣
↪needs sage.symbolic
0
```

class sage.groups.misc_gps.imaginary_groups.**ImaginaryGroup**(*base*, *category*)

Bases: `UniqueRepresentation`, `Parent`

A group whose elements are purely imaginary.

INPUT:

- `base` – a SageMath parent

- `category` – a category

EXAMPLES:

```
sage: from sage.groups.misc_gps.imaginary_groups import ImaginaryGroup
sage: J = ImaginaryGroup(ZZ)
sage: J(0)
0
sage: J(imag=100)
100*I
sage: J(3*I)                                                              #␣
↪needs sage.symbolic
3*I
sage: J(1 + 2*I)                                                          #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: 2*I + 1 is not in
Imaginary Group over Integer Ring
because it is not purely imaginary
```

```
>>> from sage.all import *
>>> from sage.groups.misc_gps.imaginary_groups import ImaginaryGroup
>>> J = ImaginaryGroup(ZZ)
>>> J(Integer(0))
0
>>> J(imag=Integer(100))
100*I
>>> J(Integer(3)*I)                                                        ␣
↪      # needs sage.symbolic
3*I
>>> J(Integer(1) + Integer(2)*I)                                           ␣
↪              # needs sage.symbolic
Traceback (most recent call last):
...
ValueError: 2*I + 1 is not in
Imaginary Group over Integer Ring
because it is not purely imaginary
```

Element

alias of *ImaginaryElement*

CHAPTER

TWENTYSEVEN

PERMUTATION GROUPS

27.1 Catalog of permutation groups

Type `groups.permutation.<tab>` to access examples of groups implemented as permutation groups.

27.2 Constructor for permutations

This module contains the generic constructor to build element of the symmetric groups (or more general permutation groups) called `PermutationGroupElement`. These objects have a more group theoretic flavor than the more combinatorial `Permutation`.

`sage.groups.perm_gps.constructor.`**`PermutationGroupElement`**(*g*, *parent=None*, *check=True*)

Build a permutation from `g`.

INPUT:

- `g` – either
  - a list of images
  - a tuple describing a single cycle
  - a list of tuples describing the cycle decomposition
  - a string describing the cycle decomposition
- `parent` – (optional) an ambient permutation group for the result; it is mandatory if you want a permutation on a domain different from $\{1, \ldots, n\}$
- `check` – boolean (default: `True`); whether additional check are performed. Setting it to `False` is likely to result in faster code.

EXAMPLES:

Initialization as a list of images:

```
sage: p = PermutationGroupElement([1,4,2,3])
sage: p
(2,4,3)
sage: p.parent()
Symmetric group of order 4! as a permutation group
```

```
>>> from sage.all import *
>>> p = PermutationGroupElement([Integer(1),Integer(4),Integer(2),Integer(3)])
>>> p
```

(continues on next page)

(continued from previous page)

```
(2,4,3)
>>> p.parent()
Symmetric group of order 4! as a permutation group
```

Initialization as a list of cycles:

```
sage: p = PermutationGroupElement([(3,5),(4,6,9)])
sage: p
(3,5)(4,6,9)
sage: p.parent()
Symmetric group of order 9! as a permutation group
```

```
>>> from sage.all import *
>>> p = PermutationGroupElement([(Integer(3),Integer(5)),(Integer(4),Integer(6),
↪Integer(9))])
>>> p
(3,5)(4,6,9)
>>> p.parent()
Symmetric group of order 9! as a permutation group
```

Initialization as a string representing a cycle decomposition:

```
sage: p = PermutationGroupElement('(2,4)(3,5)')
sage: p
(2,4)(3,5)
sage: p.parent()
Symmetric group of order 5! as a permutation group
```

```
>>> from sage.all import *
>>> p = PermutationGroupElement('(2,4)(3,5)')
>>> p
(2,4)(3,5)
>>> p.parent()
Symmetric group of order 5! as a permutation group
```

By default the constructor assumes that the domain is $\{1, \dots, n\}$ but it can be set to anything via its second `parent` argument:

```
sage: S = SymmetricGroup(['a', 'b', 'c', 'd', 'e'])
sage: PermutationGroupElement(['e', 'c', 'b', 'a', 'd'], S)
('a','e','d')('b','c')
sage: PermutationGroupElement(('a', 'b', 'c'), S)
('a','b','c')
sage: PermutationGroupElement([('a', 'c'), ('b', 'e')], S)
('a','c')('b','e')
sage: PermutationGroupElement("('a','b','e')('c','d')", S)
('a','b','e')('c','d')
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(['a', 'b', 'c', 'd', 'e'])
>>> PermutationGroupElement(['e', 'c', 'b', 'a', 'd'], S)
('a','e','d')('b','c')
```

(continues on next page)

(continued from previous page)

```
>>> PermutationGroupElement(('a', 'b', 'c'), S)
('a','b','c')
>>> PermutationGroupElement([('a', 'c'), ('b', 'e')], S)
('a','c')('b','e')
>>> PermutationGroupElement("('a','b','e')('c','d')", S)
('a','b','e')('c','d')
```

But in this situation, you might want to use the more direct:

```
sage: S(['e', 'c', 'b', 'a', 'd'])
('a','e','d')('b','c')
sage: S(('a', 'b', 'c'))
('a','b','c')
sage: S([('a', 'c'), ('b', 'e')])
('a','c')('b','e')
sage: S("('a','b','e')('c','d')")
('a','b','e')('c','d')
```

```
>>> from sage.all import *
>>> S(['e', 'c', 'b', 'a', 'd'])
('a','e','d')('b','c')
>>> S(('a', 'b', 'c'))
('a','b','c')
>>> S([('a', 'c'), ('b', 'e')])
('a','c')('b','e')
>>> S("('a','b','e')('c','d')")
('a','b','e')('c','d')
```

`sage.groups.perm_gps.constructor.`**`standardize_generator`**(*g*, *convert_dict=None*, *as_cycles=False*)

Standardize the input for permutation group elements to a list or a list of tuples.

This was factored out of the `PermutationGroupElement.__init__` since `PermutationGroup_generic.__init__` needs to do the same computation in order to compute the domain of a group when it's not explicitly specified.

INPUT:

- `g` – a `list`, `tuple`, `string`, `GapElement`, *`PermutationGroupElement`*, or `Permutation`
- `convert_dict` – (optional) a dictionary used to convert the points to a number compatible with GAP
- `as_cycles` – boolean (default: `False`); whether the output should be as cycles or in one-line notation

OUTPUT: the permutation in as a list in one-line notation or a list of cycles as tuples

EXAMPLES:

```
sage: from sage.groups.perm_gps.constructor import standardize_generator
sage: standardize_generator('(1,2)')
[2, 1]

sage: p = PermutationGroupElement([(1,2)])
sage: standardize_generator(p)
[2, 1]
sage: standardize_generator(p._gap_())
```

(continues on next page)

(continued from previous page)

```
[2, 1]
sage: standardize_generator((1,2))
[2, 1]
sage: standardize_generator([(1,2)])
[2, 1]

sage: standardize_generator(p, as_cycles=True)
[(1, 2)]
sage: standardize_generator(p._gap_(), as_cycles=True)
[(1, 2)]
sage: standardize_generator((1,2), as_cycles=True)
[(1, 2)]
sage: standardize_generator([(1,2)], as_cycles=True)
[(1, 2)]

sage: standardize_generator(Permutation([2,1,3]))
[2, 1, 3]
sage: standardize_generator(Permutation([2,1,3]), as_cycles=True)
[(1, 2), (3,)]
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.constructor import standardize_generator
>>> standardize_generator('(1,2)')
[2, 1]

>>> p = PermutationGroupElement([(Integer(1),Integer(2))])
>>> standardize_generator(p)
[2, 1]
>>> standardize_generator(p._gap_())
[2, 1]
>>> standardize_generator((Integer(1),Integer(2)))
[2, 1]
>>> standardize_generator([(Integer(1),Integer(2))])
[2, 1]

>>> standardize_generator(p, as_cycles=True)
[(1, 2)]
>>> standardize_generator(p._gap_(), as_cycles=True)
[(1, 2)]
>>> standardize_generator((Integer(1),Integer(2)), as_cycles=True)
[(1, 2)]
>>> standardize_generator([(Integer(1),Integer(2))], as_cycles=True)
[(1, 2)]

>>> standardize_generator(Permutation([Integer(2),Integer(1),Integer(3)]))
[2, 1, 3]
>>> standardize_generator(Permutation([Integer(2),Integer(1),Integer(3)]), as_
↪cycles=True)
[(1, 2), (3,)]
```

```
sage: d = {'a': 1, 'b': 2}
```

(continues on next page)

(continued from previous page)

```
sage: p = SymmetricGroup(['a', 'b']).gen(0); p
('a','b')
sage: standardize_generator(p, convert_dict=d)
[2, 1]
sage: standardize_generator(p._gap_(), convert_dict=d)
[2, 1]
sage: standardize_generator(('a','b'), convert_dict=d)
[2, 1]
sage: standardize_generator([('a','b')], convert_dict=d)
[2, 1]

sage: standardize_generator(p, convert_dict=d, as_cycles=True)
[(1, 2)]
sage: standardize_generator(p._gap_(), convert_dict=d, as_cycles=True)
[(1, 2)]
sage: standardize_generator(('a','b'), convert_dict=d, as_cycles=True)
[(1, 2)]
sage: standardize_generator([('a','b')], convert_dict=d, as_cycles=True)
[(1, 2)]
```

```
>>> from sage.all import *
>>> d = {'a': Integer(1), 'b': Integer(2)}
>>> p = SymmetricGroup(['a', 'b']).gen(Integer(0)); p
('a','b')
>>> standardize_generator(p, convert_dict=d)
[2, 1]
>>> standardize_generator(p._gap_(), convert_dict=d)
[2, 1]
>>> standardize_generator(('a','b'), convert_dict=d)
[2, 1]
>>> standardize_generator([('a','b')], convert_dict=d)
[2, 1]

>>> standardize_generator(p, convert_dict=d, as_cycles=True)
[(1, 2)]
>>> standardize_generator(p._gap_(), convert_dict=d, as_cycles=True)
[(1, 2)]
>>> standardize_generator(('a','b'), convert_dict=d, as_cycles=True)
[(1, 2)]
>>> standardize_generator([('a','b')], convert_dict=d, as_cycles=True)
[(1, 2)]
```

sage.groups.perm_gps.constructor.**string_to_tuples**(*g*)

EXAMPLES:

```
sage: from sage.groups.perm_gps.constructor import string_to_tuples
sage: string_to_tuples('(1,2,3)')
[(1, 2, 3)]
sage: string_to_tuples('(1,2,3)(4,5)')
[(1, 2, 3), (4, 5)]
sage: string_to_tuples(' (1,2, 3) (4,5)')
[(1, 2, 3), (4, 5)]
```

(continues on next page)

(continued from previous page)

```
sage: string_to_tuples('(1,2)(3)')
[(1, 2), (3,)]
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.constructor import string_to_tuples
>>> string_to_tuples('(1,2,3)')
[(1, 2, 3)]
>>> string_to_tuples('(1,2,3)(4,5)')
[(1, 2, 3), (4, 5)]
>>> string_to_tuples(' (1,2, 3) (4,5)')
[(1, 2, 3), (4, 5)]
>>> string_to_tuples('(1,2)(3)')
[(1, 2), (3,)]
```

27.3 Permutation groups

A permutation group is a finite group G whose elements are permutations of a given finite set X (i.e., bijections $X \longrightarrow X$) and whose group operation is the composition of permutations. The number of elements of X is called the degree of G.

In Sage, a permutation is represented as either a string that defines a permutation using disjoint cycle notation, or a list of tuples, which represent disjoint cycles. That is:

```
(a,...,b)(c,...,d)...(e,...,f)  <--> [(a,...,b), (c,...,d),..., (e,...,f)]
                    () = identity <--> []
```

You can make the "named" permutation groups (see `permgp_named.py`) and use the following constructions:

- permutation group generated by elements,
- `direct_product_permgroups`, which takes a list of permutation groups and returns their direct product.

JOKE: Q: What's hot, chunky, and acts on a polygon? A: Dihedral soup. Renteln, P. and Dundes, A. "Foolproof: A Sampling of Mathematical Folk Humor." Notices Amer. Math. Soc. 52, 24-34, 2005.

27.3.1 Index of methods

Here are the methods of a *PermutationGroup()*

are_conjugate()	Return whether `H1` and `H2` are conjugate subgroups in `G`.
as_finitely_presented_group()	Return a finitely presented group isomorphic to `self`.
blocks_all()	Return the list of block systems of imprimitivity.
cardinality()	Return the number of elements of this group.
center()	Return the subgroup of elements that commute with every element of this group.
centralizer()	Return the centralizer of `g` in `self`.
character()	Return a group character from `values`, where `values` is a list of the values of the character evaluated on the conjugacy classes.
character_table()	Return the matrix of values of the irreducible characters of a permutation group G at the conjugacy classes of G.
cohomology()	Compute the group cohomology $H^n(G, F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime.

continues on next page

Table 1 – continued from previous page

cohomology_part()	Compute the p-part of the group cohomology $H^n(G,F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime.
commutator()	Return the commutator subgroup of a group, or of a pair of groups.
composition_series()	Return the composition series of this group as a list of permutation groups.
conjugacy_class()	Return the conjugacy class of `g` inside the group `self`.
conjugacy_classes()	Return a list with all the conjugacy classes of `self`.
conjugacy_classes_representatives()	Return a complete list of representatives of conjugacy classes in a permutation group G.
conjugacy_classes_subgroups()	Return a complete list of representatives of conjugacy classes of subgroups in a permutation group G.
conjugate()	Return the group formed by conjugating `self` with `g`.
construction()	Return the construction of `self`.
cosets()	Return a list of the cosets of `S` in `self`.
degree()	Return the degree of this permutation group.
derived_series()	Return the derived series of this group as a list of permutation groups.
direct_product()	Wraps GAP's `DirectProduct`, `Embedding`, and `Projection`.
domain()	Return the underlying set that this permutation group acts on.
exponent()	Compute the exponent of the group.
fitting_subgroup()	Return the Fitting subgroup of `self`.
fixed_points()	Return the list of points fixed by `self`, i.e., the subset of `.domain()` not moved by any element of `self`.
frattini_subgroup()	Return the Frattini subgroup of `self`.
gen()	Return the i-th generator of `self`; that is, the i-th element of the list `self.gens()`.
gens()	Return tuple of generators of this group.
gens_small()	For this group, returns a generating set which has few elements.
group_id()	Return the ID code of this group, which is a list of two integers.
group_primitive_id()	Return the index of this group in the GAP database of primitive groups.
holomorph()	The holomorph of a group as a permutation group.
homology()	Compute the group homology $H_n(G,F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime. Wraps HAP's `GroupHomology` function, written by Graham Ellis.
homology_part()	Compute the p-part of the group homology $H_n(G,F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime. Wraps HAP's `Homology` function, written by Graham Ellis, applied to the p-Sylow subgroup of G.
id()	Return the ID code of this group, which is a list of two integers.
intersection()	Return the permutation group that is the intersection of `self` and `other`.
irreducible_characters()	Return a list of the irreducible characters of `self`.
is_cyclic()	Return `True` if this group is cyclic.
is_elementary_abelian()	Return `True` if this group is elementary abelian. An elementary abelian group is a finite abelian group, where every nontrivial element has order p, where p is a prime.
is_isomorphic()	Return `True` if the groups are isomorphic.
is_monomial()	Return `True` if the group is monomial. A finite group is monomial if every irreducible complex character is induced from a linear character of a subgroup.
is_nilpotent()	Return `True` if this group is nilpotent.
is_normal()	Return `True` if this group is a normal subgroup of `other`.
is_perfect()	Return `True` if this group is perfect. A group is perfect if it equals its derived subgroup.
is_pgroup()	Return `True` if this group is a p-group.

continues on next page

Table 1 – continued from previous page

is_polycyclic()	Return True if this group is polycyclic. A group is polycyclic if it has a subnormal series with cyclic factors. (For finite groups, this is the same as if the group is solvable - see is_solvable().)
is_primitive()	Return True if self acts primitively on domain.
is_regular()	Return True if self acts regularly on domain.
is_semi_regular()	Return True if self acts semi-regularly on domain.
is_simple()	Return True if the group is simple.
is_solvable()	Return True if the group is solvable.
is_subgroup()	Return True if self is a subgroup of other.
is_supersolvable()	Return True if the group is supersolvable.
is_transitive()	Return True if self acts transitively on domain.
isomorphism_to()	Return an isomorphism from self to right if the groups are isomorphic, otherwise None.
isomorphism_type_info_simple_group()	If the group is simple, then this returns the name of the group.
iteration()	Return an iterator over the elements of this group.
largest_moved_point()	Return the largest point moved by a permutation in this group.
list()	Return list of all elements of this group.
lower_central_series()	Return the lower central series of this group as a list of permutation groups.
maximal_normal_subgroups()	Return the maximal proper normal subgroups of self.
minimal_generating_set()	Return a minimal generating set.
minimal_normal_subgroups()	Return the nontrivial minimal normal subgroups self.
molien_series()	Return the Molien series of a permutation group. The function
ngens()	Return the number of generators of self.
non_fixed_points()	Return the list of points not fixed by self, i.e., the subset of self.domain() moved by some element of self.
normal_subgroups()	Return the normal subgroups of this group as a (sorted in increasing order) list of permutation groups.
normalizer()	Return the normalizer of g in self.
normalizes()	Return True if the group other is normalized by self.
poincare_series()	Return the Poincaré series of $G \mod p$ ($p \geq 2$ must be a prime), for n large.
random_element()	Return a random element of this group.
representative_action()	Return an element of self that maps x to y if it exists.
semidirect_product()	The semidirect product of self with N.
sign_representation()	Return the sign representation of self over base_ring.
socle()	Return the socle of self.
solvable_radical()	Return the solvable radical of self.
stabilizer()	Return the subgroup of self which stabilize the given position. self and its stabilizers must have same degree.
strong_generating_system()	Return a Strong Generating System of self according the given base for the right action of self on itself.
structure_description()	Return a string that tries to describe the structure of G.
subgroup()	Wraps the PermutationGroup_subgroup constructor. The argument gens is a list of elements of self.
subgroups()	Return a list of all the subgroups of self.
sylow_subgroup()	Return a Sylow p-subgroup of the finite group G, where p is a prime.

continues on next page

Table 1 – continued from previous page

`transversals()`	If G is a permutation group acting on the set $X = \{1, 2, ..., n\}$ and H is the stabilizer subgroup of <integer>, a right (respectively left) transversal is a set containing exactly one element from each right (respectively left) coset of H. This method returns a right transversal of `self` by the stabilizer of `self` on <integer> position.
`trivial_character()`	Return the trivial character of `self`.
`upper_central_series()`	Return the upper central series of this group as a list of permutation groups.

AUTHORS:

- David Joyner (2005-10-14): first version
- David Joyner (2005-11-17)
- William Stein (2005-11-26): rewrite to better wrap Gap
- David Joyner (2005-12-21)
- William Stein and David Joyner (2006-01-04): added conjugacy_class_representatives
- David Joyner (2006-03): reorganization into subdirectory perm_gps; added __contains__, has_element; fixed _cmp_; added subgroup class+methods, PGL,PSL,PSp, PSU classes,
- David Joyner (2006-06): added PGU, functionality to SymmetricGroup, AlternatingGroup, direct_product_permgroups
- David Joyner (2006-08): added degree, ramification_module_decomposition_modular_curve and ramification_module_decomposition_hurwitz_curve methods to PSL(2,q), MathieuGroup, is_isomorphic
- Bobby Moretti (2006)-10): Added KleinFourGroup, fixed bug in DihedralGroup
- David Joyner (2006-10): added is_subgroup (fixing a bug found by Kiran Kedlaya), is_solvable, normalizer, is_normal_subgroup, Suzuki
- David Kohel (2007-02): fixed __contains__ to not enumerate group elements, following the convention for __call__
- David Harvey, Mike Hansen, Nick Alexander, William Stein (2007-02,03,04,05): Various patches
- Nathan Dunfield (2007-05): added orbits
- David Joyner (2007-06): added subgroup method (suggested by David Kohel), composition_series, lower_central_series, upper_central_series, cayley_table, quotient_group, sylow_subgroup, is_cyclic, homology, homology_part, cohomology, cohomology_part, poincare_series, molien_series, is_simple, is_monomial, is_supersolvable, is_nilpotent, is_perfect, is_polycyclic, is_elementary_abelian, is_pgroup, gens_small, isomorphism_type_info_simple_group. moved all the"named" groups to a new file.
- Nick Alexander (2007-07): move is_isomorphic to isomorphism_to, add from_gap_list
- William Stein (2007-07): put is_isomorphic back (and make it better)
- David Joyner (2007-08): fixed bugs in composition_series, upper/lower_central_series, derived_series,
- David Joyner (2008-06): modified is_normal (reported by W. J. Palenstijn), and added normalizes
- David Joyner (2008-08): Added example to docstring of cohomology.
- Simon King (2009-04): __cmp__ methods for PermutationGroup_generic and PermutationGroup_subgroup
- Nicolas Borie (2009): Added orbit, transversals, stabiliser and strong_generating_system methods
- Christopher Swenson (2012): Added a special case to compute the order efficiently. (This patch Copyright 2012 Google Inc. All Rights Reserved.)
- Javier Lopez Pena (2013): Added conjugacy classes.

- Sebastian Oehms (2018): added _coerce_map_from_ in order to use isomorphism coming up with as_permutation_group method (Issue #25706)
- Christian Stump (2018): Added alternative implementation of strong_generating_system directly using GAP.
- Sebastian Oehms (2018): Added `PermutationGroup_generic._Hom_()` to use `sage.groups.libgap_morphism.GroupHomset_libgap` and `PermutationGroup_generic.gap()` and `PermutationGroup_generic._subgroup_constructor()` (for compatibility to libgap framework, see Issue #26750

REFERENCES:

- Cameron, P., Permutation Groups. New York: Cambridge University Press, 1999.
- Wielandt, H., Finite Permutation Groups. New York: Academic Press, 1964.
- Dixon, J. and Mortimer, B., Permutation Groups, Springer-Verlag, Berlin/New York, 1996.

Note

Though Suzuki groups are okay, Ree groups should *not* be wrapped as permutation groups - the construction is too slow - unless (for small values or the parameter) they are made using explicit generators.

`sage.groups.perm_gps.permgroup.`**`PermutationGroup`**`(`*gens=None*, **args*, ***kwds*`)`

Return the permutation group associated to x (typically a list of generators).

INPUT:

- `gens` – (default: `None`) list of generators
- `gap_group` – (optional) a gap permutation group
- `canonicalize` – boolean (default: `True`); if `True`, sort generators and remove duplicates

OUTPUT: a permutation group

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G
Permutation Group with generators [(3,4), (1,2,3)(4,5)]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))],[(Integer(3),Integer(4))]])
>>> G
Permutation Group with generators [(3,4), (1,2,3)(4,5)]
```

We can also make permutation groups from PARI groups:

```
sage: # needs sage.libs.pari
sage: H = pari('x^4 - 2*x^3 - 2*x + 1').polgalois()
sage: G = PariGroup(H, 4); G
PARI group [8, -1, 3, "D(4)"] of degree 4
sage: H = PermutationGroup(G); H
Transitive group number 3 of degree 4
sage: H.gens()
((1,2,3,4), (1,3))
```

```
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> H = pari('x^4 - 2*x^3 - 2*x + 1').polgalois()
>>> G = PariGroup(H, Integer(4)); G
PARI group [8, -1, 3, "D(4)"] of degree 4
>>> H = PermutationGroup(G); H
Transitive group number 3 of degree 4
>>> H.gens()
((1,2,3,4), (1,3))
```

We can also create permutation groups whose generators are GAP permutation objects:

```
sage: p = gap('(1,2)(3,7)(4,6)(5,8)'); p
(1,2)(3,7)(4,6)(5,8)
sage: PermutationGroup([p])
Permutation Group with generators [(1,2)(3,7)(4,6)(5,8)]
```

```
>>> from sage.all import *
>>> p = gap('(1,2)(3,7)(4,6)(5,8)'); p
(1,2)(3,7)(4,6)(5,8)
>>> PermutationGroup([p])
Permutation Group with generators [(1,2)(3,7)(4,6)(5,8)]
```

Permutation groups can work on any domain. In the following examples, the permutations are specified in list notation, according to the order of the elements of the domain:

```
sage: list(PermutationGroup([['b','c','a']], domain=['a','b','c']))
[(), ('a','b','c'), ('a','c','b')]
sage: list(PermutationGroup([['b','c','a']], domain=['b','c','a']))
[()]
sage: list(PermutationGroup([['b','c','a']], domain=['a','c','b']))
[(), ('a','b')]
```

```
>>> from sage.all import *
>>> list(PermutationGroup([['b','c','a']], domain=['a','b','c']))
[(), ('a','b','c'), ('a','c','b')]
>>> list(PermutationGroup([['b','c','a']], domain=['b','c','a']))
[()]
>>> list(PermutationGroup([['b','c','a']], domain=['a','c','b']))
[(), ('a','b')]
```

There is an underlying gap object that implements each permutation group:

```
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: G._gap_()
Group( [ (1,2,3,4) ] )
sage: gap(G)
Group( [ (1,2,3,4) ] )
sage: gap(G) is G._gap_()
True
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: current_randstate().set_seed_gap()
sage: G1, G2 = G._gap_().DerivedSeries()
```

(continues on next page)

(continued from previous page)

```
sage: G1
Group( [ (3,4), (1,2,3)(4,5) ] )
sage: G2.GeneratorsSmallest()
[ (3,4,5), (2,3)(4,5), (1,2)(4,5) ]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> G._gap_()
Group( [ (1,2,3,4) ] )
>>> gap(G)
Group( [ (1,2,3,4) ] )
>>> gap(G) is G._gap_()
True
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))],[(Integer(3),Integer(4))]])
>>> current_randstate().set_seed_gap()
>>> G1, G2 = G._gap_().DerivedSeries()
>>> G1
Group( [ (3,4), (1,2,3)(4,5) ] )
>>> G2.GeneratorsSmallest()
[ (3,4,5), (2,3)(4,5), (1,2)(4,5) ]
```

We can create a permutation group from a group action:

```
sage: a = lambda x: (2*x) % 7
sage: H = PermutationGroup(action=a, domain=range(7))                          #␣
↪needs sage.combinat
sage: H.orbits()                                                               #␣
↪needs sage.libs.pari
((0,), (1, 2, 4), (3, 6, 5))
sage: H.gens()                                                                 #␣
↪needs sage.libs.pari
((1,2,4), (3,6,5))
```

```
>>> from sage.all import *
>>> a = lambda x: (Integer(2)*x) % Integer(7)
>>> H = PermutationGroup(action=a, domain=range(Integer(7)))                   ␣
↪     # needs sage.combinat
>>> H.orbits()                                                                #␣
↪needs sage.libs.pari
((0,), (1, 2, 4), (3, 6, 5))
>>> H.gens()                                                                  #␣
↪needs sage.libs.pari
((1,2,4), (3,6,5))
```

Note that we provide generators for the acting group. The permutation group we construct is its homomorphic image:

```
sage: # needs sage.combinat
sage: a = lambda g, x: vector(g*x, immutable=True)
sage: X = [vector(x, immutable=True) for x in GF(3)^2]
sage: G = SL(2,3); G.gens()
```

(continues on next page)

(continued from previous page)

```
(
[1 1]  [0 1]
[0 1], [2 0]
)
sage: H = PermutationGroup(G.gens(), action=a, domain=X)
sage: H.orbits()
(((0, 0),), ((1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2)))
sage: H.gens()
(((0,1),(1,1),(2,1))((0,2),(2,2),(1,2)),
 ((1,0),(0,2),(2,0),(0,1))((1,1),(1,2),(2,2),(2,1)))
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> a = lambda g, x: vector(g*x, immutable=True)
>>> X = [vector(x, immutable=True) for x in GF(Integer(3))**Integer(2)]
>>> G = SL(Integer(2),Integer(3)); G.gens()
(
[1 1]  [0 1]
[0 1], [2 0]
)
>>> H = PermutationGroup(G.gens(), action=a, domain=X)
>>> H.orbits()
(((0, 0),), ((1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2)))
>>> H.gens()
(((0,1),(1,1),(2,1))((0,2),(2,2),(1,2)),
 ((1,0),(0,2),(2,0),(0,1))((1,1),(1,2),(2,2),(2,1)))
```

The orbits of the conjugation action are the conjugacy classes, i.e., in bijection with integer partitions:

```
sage: a = lambda g, x: g*x*g^-1
sage: [len(PermutationGroup(SymmetricGroup(n).gens(), action=a,              #␣
↪needs sage.combinat
....:                          domain=SymmetricGroup(n)).orbits())
....:  for n in range(1, 8)]
[1, 2, 3, 5, 7, 11, 15]
```

```
>>> from sage.all import *
>>> a = lambda g, x: g*x*g**-Integer(1)
>>> [len(PermutationGroup(SymmetricGroup(n).gens(), action=a,              #␣
↪needs sage.combinat
...                          domain=SymmetricGroup(n)).orbits())
...  for n in range(Integer(1), Integer(8))]
[1, 2, 3, 5, 7, 11, 15]
```

class sage.groups.perm_gps.permgroup.**PermutationGroup_action**(*gens*, *action*, *domain*, *gap_group=None*, *category=None*, *canonicalize=None*)

Bases: *PermutationGroup_generic*

A permutation group given by a finite group action.

EXAMPLES:

A cyclic action:

```
sage: n = 3
sage: a = lambda x: SetPartition([[e % n + 1 for e in b] for b in x])
sage: S = SetPartitions(n)                                              #␣
↪needs sage.combinat
sage: G = PermutationGroup(action=a, domain=S)                          #␣
↪needs sage.combinat
sage: G.orbits()                                                        #␣
↪needs sage.combinat
(({{1}, {2}, {3}},),
 ({{1, 2}, {3}}, {{1}, {2, 3}}, {{1, 3}, {2}}),
 ({{1, 2, 3}},))
```

```
>>> from sage.all import *
>>> n = Integer(3)
>>> a = lambda x: SetPartition([[e % n + Integer(1) for e in b] for b in x])
>>> S = SetPartitions(n)                                              #␣
↪needs sage.combinat
>>> G = PermutationGroup(action=a, domain=S)                          #␣
↪needs sage.combinat
>>> G.orbits()                                                        #␣
↪needs sage.combinat
(({{1}, {2}, {3}},),
 ({{1, 2}, {3}}, {{1}, {2, 3}}, {{1, 3}, {2}}),
 ({{1, 2, 3}},))
```

The regular action of the symmetric group:

```
sage: a = lambda g, x: g*x*g^-1
sage: S = SymmetricGroup(3)
sage: G = PermutationGroup(S.gens(), action=a, domain=S)                #␣
↪needs sage.combinat
sage: G.orbits()                                                        #␣
↪needs sage.combinat
(((),), ((1,3,2), (1,2,3)), ((2,3), (1,3), (1,2)))
```

```
>>> from sage.all import *
>>> a = lambda g, x: g*x*g**-Integer(1)
>>> S = SymmetricGroup(Integer(3))
>>> G = PermutationGroup(S.gens(), action=a, domain=S)                #␣
↪needs sage.combinat
>>> G.orbits()                                                        #␣
↪needs sage.combinat
(((),), ((1,3,2), (1,2,3)), ((2,3), (1,3), (1,2)))
```

The trivial action of the symmetric group:

```
sage: PermutationGroup(SymmetricGroup(3).gens(),                        #␣
↪needs sage.combinat
....:                  action=lambda g, x: x, domain=[1])
Permutation Group with generators [()]
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> PermutationGroup(SymmetricGroup(Integer(3)).gens(),
↪     # needs sage.combinat
...                      action=lambda g, x: x, domain=[Integer(1)])
Permutation Group with generators [()]
```

orbits()

Return the orbits of the elements of the domain under the default group action.

EXAMPLES:

```
sage: a = lambda x: (2*x) % 7
sage: G = PermutationGroup(action=a, domain=range(7))                     #
↪needs sage.combinat
sage: G.orbits()                                                          #
↪needs sage.combinat
((0,), (1, 2, 4), (3, 6, 5))
```

```
>>> from sage.all import *
>>> a = lambda x: (Integer(2)*x) % Integer(7)
>>> G = PermutationGroup(action=a, domain=range(Integer(7)))
↪     # needs sage.combinat
>>> G.orbits()                                                          #
↪needs sage.combinat
((0,), (1, 2, 4), (3, 6, 5))
```

class sage.groups.perm_gps.permgroup.**PermutationGroup_generic**(*gens=None*, *gap_group=None*, *canonicalize=True*, *domain=None*, *category=None*)

Bases: *FiniteGroup*

A generic permutation group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G
Permutation Group with generators [(3,4), (1,2,3)(4,5)]
sage: G.center()
Subgroup generated by [()] of
 (Permutation Group with generators [(3,4), (1,2,3)(4,5)])
sage: G.group_id()
[120, 34]
sage: n = G.order(); n
120
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: TestSuite(G).run()
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))],[(Integer(3),Integer(4))]])
>>> G
Permutation Group with generators [(3,4), (1,2,3)(4,5)]
>>> G.center()
```

(continues on next page)

(continued from previous page)

```
Subgroup generated by [()] of
 (Permutation Group with generators [(3,4), (1,2,3)(4,5)])
>>> G.group_id()
[120, 34]
>>> n = G.order(); n
120
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))],[(Integer(3),Integer(4))]])
>>> TestSuite(G).run()
```

Element

alias of *PermutationGroupElement*

Subgroup

alias of *PermutationGroup_subgroup*

are_conjugate(*H1*, *H2*)

Return whether H1 and H2 are conjugate subgroups in G.

EXAMPLES:

```
sage: G = SymmetricGroup(3)
sage: H1 = PermutationGroup([(1,2)])
sage: H2 = PermutationGroup([(2,3)])
sage: G.are_conjugate(H1, H2)
True
sage: G = SymmetricGroup(4)
sage: H1 = PermutationGroup([[(1,3),(2,4)], [(1,2),(3,4)]])
sage: H2 = PermutationGroup([[(1,2)], [(1,2),(3,4)]])
sage: G.are_conjugate(H1, H2)
False
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> H1 = PermutationGroup([(Integer(1),Integer(2))])
>>> H2 = PermutationGroup([(Integer(2),Integer(3))])
>>> G.are_conjugate(H1, H2)
True
>>> G = SymmetricGroup(Integer(4))
>>> H1 = PermutationGroup([[(Integer(1),Integer(3)),(Integer(2),Integer(4))],
↪[(Integer(1),Integer(2)),(Integer(3),Integer(4))]])
>>> H2 = PermutationGroup([[(Integer(1),Integer(2))], [(Integer(1),
↪Integer(2)),(Integer(3),Integer(4))]])
>>> G.are_conjugate(H1, H2)
False
```

as_finitely_presented_group(*reduced=False*)

Return a finitely presented group isomorphic to self.

This method acts as wrapper for the GAP function IsomorphismFpGroupByGenerators, which yields an isomorphism from a given group to a finitely presented group.

INPUT:

- reduced – (default: False) if True, *FinitelyPresentedGroup.simplified* is called, attempting to simplify the presentation of the finitely presented group to be returned

OUTPUT: finite presentation of self, obtained by taking the image of the isomorphism returned by the GAP function IsomorphismFpGroupByGenerators

ALGORITHM: uses GAP

EXAMPLES:

```
sage: CyclicPermutationGroup(50).as_finitely_presented_group()
Finitely presented group < a | a^50 >
sage: DihedralGroup(4).as_finitely_presented_group()
Finitely presented group < a, b | b^2, a^4, (b*a)^2 >
sage: GeneralDihedralGroup([2,2]).as_finitely_presented_group()
Finitely presented group < a, b, c | a^2, b^2, c^2, (c*b)^2, (c*a)^2, (b*a)^2␣
↪>
```

```
>>> from sage.all import *
>>> CyclicPermutationGroup(Integer(50)).as_finitely_presented_group()
Finitely presented group < a | a^50 >
>>> DihedralGroup(Integer(4)).as_finitely_presented_group()
Finitely presented group < a, b | b^2, a^4, (b*a)^2 >
>>> GeneralDihedralGroup([Integer(2),Integer(2)]).as_finitely_presented_
↪group()
Finitely presented group < a, b, c | a^2, b^2, c^2, (c*b)^2, (c*a)^2, (b*a)^2␣
↪>
```

GAP algorithm is not guaranteed to produce minimal or canonical presentation:

```
sage: G = PermutationGroup(['(1,2,3,4,5)', '(1,5)(2,4)'])
sage: G.is_isomorphic(DihedralGroup(5))
True
sage: K = G.as_finitely_presented_group(); K
Finitely presented group < a, b | b^2, (b*a)^2, b*a^-3*b*a^2 >
sage: K.as_permutation_group().is_isomorphic(DihedralGroup(5))
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3,4,5)', '(1,5)(2,4)'])
>>> G.is_isomorphic(DihedralGroup(Integer(5)))
True
>>> K = G.as_finitely_presented_group(); K
Finitely presented group < a, b | b^2, (b*a)^2, b*a^-3*b*a^2 >
>>> K.as_permutation_group().is_isomorphic(DihedralGroup(Integer(5)))
True
```

We can attempt to reduce the output presentation:

```
sage: H = PermutationGroup(['(1,2,3,4,5)', '(1,3,5,2,4)'])
sage: H.as_finitely_presented_group()
Finitely presented group < a, b | b^-2*a^-1, b*a^-2 >
sage: H.as_finitely_presented_group(reduced=True)
Finitely presented group < a | a^5 >
```

```
>>> from sage.all import *
>>> H = PermutationGroup(['(1,2,3,4,5)', '(1,3,5,2,4)'])
>>> H.as_finitely_presented_group()
Finitely presented group < a, b | b^-2*a^-1, b*a^-2 >
>>> H.as_finitely_presented_group(reduced=True)
Finitely presented group < a | a^5 >
```

AUTHORS:

- Davis Shurbert (2013-06-21): initial version

base(*seed=None*)

Return a (minimum) base of this permutation group.

A base B of a permutation group is a subset of the domain of the group such that the only group element stabilizing all of B is the identity.

INPUT:

- `seed` – (default: `None`) if given must be a subset of the domain of a base. When used, an attempt to create a base containing all or part of `seed` will be made.

EXAMPLES:

```
sage: G = PermutationGroup([(1,2,3),(6,7,8)])
sage: G.base()
[1, 6]
sage: G.base([2])
[2, 6]

sage: H = PermutationGroup([('a','b','c'),('a','y')])
sage: H.base()
['a', 'b', 'c']

sage: S = SymmetricGroup(13)
sage: S.base()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

sage: S = MathieuGroup(12)
sage: S.base()
[1, 2, 3, 4, 5]
sage: S.base([1,3,5,7,9,11]) # create a base for M12 with only odd integers
[1, 3, 5, 7, 9]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([(Integer(1),Integer(2),Integer(3)),(Integer(6),
↪Integer(7),Integer(8))])
>>> G.base()
[1, 6]
>>> G.base([Integer(2)])
[2, 6]

>>> H = PermutationGroup([('a','b','c'),('a','y')])
>>> H.base()
['a', 'b', 'c']
```

(continues on next page)

(continued from previous page)

```
>>> S = SymmetricGroup(Integer(13))
>>> S.base()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

>>> S = MathieuGroup(Integer(12))
>>> S.base()
[1, 2, 3, 4, 5]
>>> S.base([Integer(1),Integer(3),Integer(5),Integer(7),Integer(9),
↪Integer(11)]) # create a base for M12 with only odd integers
[1, 3, 5, 7, 9]
```

blocks_all(*representatives=True*)

Return the list of block systems of imprimitivity.

For more information on primitivity, see the Wikipedia article on primitive group actions.

INPUT:

- `representative` – boolean; whether to return all possible block systems of imprimitivity or only one of their representatives (the block can be obtained from its representative set S by computing the orbit of S under `self`).

  This parameter is set to `True` by default (as it is GAP's default behaviour).

OUTPUT:

This method returns a description of *all* block systems. Hence, the output is a "list of lists of lists" or a "list of lists" depending on the value of `representatives`. A bit more clearly, output is:

- A list of length (#number of different block systems) of
  - block systems, each of them being defined as
    - If `representatives=True`: a list of representatives of each set of the block system
    - If `representatives=False`: a partition of the elements defining an imprimitivity block.

See also

- *is_primitive()*

EXAMPLES:

Picking an interesting group:

```
sage: # needs sage.graphs
sage: g = graphs.DodecahedralGraph()
sage: g.is_vertex_transitive()
True
sage: ag = g.automorphism_group()
sage: ag.is_primitive()
False
```

```
>>> from sage.all import *
>>> # needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
>>> g = graphs.DodecahedralGraph()
>>> g.is_vertex_transitive()
True
>>> ag = g.automorphism_group()
>>> ag.is_primitive()
False
```

Computing its blocks representatives:

```
sage: ag.blocks_all()                                                       #␣
↪needs sage.graphs
[[0, 15]]
```

```
>>> from sage.all import *
>>> ag.blocks_all()                                                       #␣
↪needs sage.graphs
[[0, 15]]
```

Now the full block:

```
sage: sorted(ag.blocks_all(representatives=False)[0])                       #␣
↪needs sage.graphs
[[0, 15], [1, 16], [2, 12], [3, 13], [4, 9],
 [5, 10], [6, 11], [7, 18], [8, 17], [14, 19]]
```

```
>>> from sage.all import *
>>> sorted(ag.blocks_all(representatives=False)[Integer(0)])               ␣
↪     # needs sage.graphs
[[0, 15], [1, 16], [2, 12], [3, 13], [4, 9],
 [5, 10], [6, 11], [7, 18], [8, 17], [14, 19]]
```

cardinality()

Return the number of elements of this group.

See also: *degree()*.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: G.order()
12
sage: G = PermutationGroup([()])
sage: G.order()
1
sage: G = PermutationGroup([])
sage: G.order()
1
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))], [(Integer(1),Integer(2))]])
>>> G.order()
```

(continues on next page)

(continued from previous page)

```
12
>>> G = PermutationGroup([()])
>>> G.order()
1
>>> G = PermutationGroup([])
>>> G.order()
1
```

`cardinality()` is just an alias:

```
sage: PermutationGroup([(1,2,3)]).cardinality()
3
```

```
>>> from sage.all import *
>>> PermutationGroup([(Integer(1),Integer(2),Integer(3))]).cardinality()
3
```

center()

Return the subgroup of elements that commute with every element of this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: G.center()
Subgroup generated by [(1,2,3,4)] of
 (Permutation Group with generators [(1,2,3,4)])
sage: G = PermutationGroup([[(1,2,3,4)], [(1,2)]])
sage: G.center()
Subgroup generated by [()] of
 (Permutation Group with generators [(1,2), (1,2,3,4)])
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> G.center()
Subgroup generated by [(1,2,3,4)] of
 (Permutation Group with generators [(1,2,3,4)])
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))],
↪[(Integer(1),Integer(2))]])
>>> G.center()
Subgroup generated by [()] of
 (Permutation Group with generators [(1,2), (1,2,3,4)])
```

centralizer(*g*)

Return the centralizer of `g` in `self`.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: g = G([(1,3)])
sage: G.centralizer(g)
Subgroup generated by [(1,3), (2,4)] of
 (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
sage: g = G([(1,2,3,4)])
```

(continues on next page)

(continued from previous page)

```
sage: G.centralizer(g)
Subgroup generated by [(1,2,3,4)] of
 (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
sage: H = G.subgroup([G([(1,2,3,4)])])
sage: G.centralizer(H)
Subgroup generated by [(1,2,3,4)] of
 (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],
↪[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> g = G([(Integer(1),Integer(3))])
>>> G.centralizer(g)
Subgroup generated by [(1,3), (2,4)] of
 (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
>>> g = G([(Integer(1),Integer(2),Integer(3),Integer(4))])
>>> G.centralizer(g)
Subgroup generated by [(1,2,3,4)] of
 (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
>>> H = G.subgroup([G([(Integer(1),Integer(2),Integer(3),Integer(4))])])
>>> G.centralizer(H)
Subgroup generated by [(1,2,3,4)] of
 (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
```

character(*values*)

Return a group character from `values`, where `values` is a list of the values of the character evaluated on the conjugacy classes.

EXAMPLES:

```
sage: G = AlternatingGroup(4)
sage: n = len(G.conjugacy_classes_representatives())
sage: G.character([1]*n)                                                    #
↪needs sage.rings.number_field
Character of Alternating group of order 4!/2 as a permutation group
```

```
>>> from sage.all import *
>>> G = AlternatingGroup(Integer(4))
>>> n = len(G.conjugacy_classes_representatives())
>>> G.character([Integer(1)]*n)
↪     # needs sage.rings.number_field
Character of Alternating group of order 4!/2 as a permutation group
```

character_table()

Return the matrix of values of the irreducible characters of a permutation group G at the conjugacy classes of G.

The columns represent the conjugacy classes of G and the rows represent the different irreducible characters in the ordering given by GAP.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3)]])
sage: G.order()
12
sage: G.character_table()                                                    #␣
↪needs sage.rings.number_field
[         1          1          1          1]
[         1 -zeta3 - 1      zeta3          1]
[         1      zeta3 -zeta3 - 1          1]
[         3          0          0         -1]
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3)]])
sage: CT = gap(G).CharacterTable()
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],␣
↪[(Integer(1),Integer(2),Integer(3))]])
>>> G.order()
12
>>> G.character_table()                                                    #␣
↪needs sage.rings.number_field
[         1          1          1          1]
[         1 -zeta3 - 1      zeta3          1]
[         1      zeta3 -zeta3 - 1          1]
[         3          0          0         -1]
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],␣
↪[(Integer(1),Integer(2),Integer(3))]])
>>> CT = gap(G).CharacterTable()
```

Type `CT.Display()` to display this nicely.

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: G.order()
8
sage: G.character_table()                                                    #␣
↪needs sage.rings.number_field
[ 1  1  1  1  1]
[ 1 -1 -1  1  1]
[ 1 -1  1 -1  1]
[ 1  1 -1 -1  1]
[ 2  0  0  0 -2]
sage: CT = gap(G).CharacterTable()
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],␣
↪[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> G.order()
8
>>> G.character_table()                                                    #␣
↪needs sage.rings.number_field
[ 1  1  1  1  1]
[ 1 -1 -1  1  1]
[ 1 -1  1 -1  1]
[ 1  1 -1 -1  1]
```

(continues on next page)

(continued from previous page)

```
[ 2  0  0  0 -2]
>>> CT = gap(G).CharacterTable()
```

Again, type `CT.Display()` to display this nicely.

```
sage: # needs sage.rings.number_field
sage: SymmetricGroup(2).character_table()
[ 1 -1]
[ 1  1]
sage: SymmetricGroup(3).character_table()
[ 1 -1  1]
[ 2  0 -1]
[ 1  1  1]
sage: SymmetricGroup(5).character_table()
[ 1 -1  1  1 -1 -1  1]
[ 4 -2  0  1  1  0 -1]
[ 5 -1  1 -1 -1  1  0]
[ 6  0 -2  0  0  0  1]
[ 5  1  1 -1  1 -1  0]
[ 4  2  0  1 -1  0 -1]
[ 1  1  1  1  1  1  1]
sage: list(AlternatingGroup(6).character_table())
[(1, 1, 1, 1, 1, 1, 1), (5, 1, 2, -1, -1, 0, 0), (5, 1, -1, 2, -1, 0, 0),
 (8, 0, -1, -1, 0, zeta5^3 + zeta5^2 + 1, -zeta5^3 - zeta5^2),
 (8, 0, -1, -1, 0, -zeta5^3 - zeta5^2, zeta5^3 + zeta5^2 + 1),
 (9, 1, 0, 0, 1, -1, -1), (10, -2, 1, 1, 0, 0, 0)]
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> SymmetricGroup(Integer(2)).character_table()
[ 1 -1]
[ 1  1]
>>> SymmetricGroup(Integer(3)).character_table()
[ 1 -1  1]
[ 2  0 -1]
[ 1  1  1]
>>> SymmetricGroup(Integer(5)).character_table()
[ 1 -1  1  1 -1 -1  1]
[ 4 -2  0  1  1  0 -1]
[ 5 -1  1 -1 -1  1  0]
[ 6  0 -2  0  0  0  1]
[ 5  1  1 -1  1 -1  0]
[ 4  2  0  1 -1  0 -1]
[ 1  1  1  1  1  1  1]
>>> list(AlternatingGroup(Integer(6)).character_table())
[(1, 1, 1, 1, 1, 1, 1), (5, 1, 2, -1, -1, 0, 0), (5, 1, -1, 2, -1, 0, 0),
 (8, 0, -1, -1, 0, zeta5^3 + zeta5^2 + 1, -zeta5^3 - zeta5^2),
 (8, 0, -1, -1, 0, -zeta5^3 - zeta5^2, zeta5^3 + zeta5^2 + 1),
 (9, 1, 0, 0, 1, -1, -1), (10, -2, 1, 1, 0, 0, 0)]
```

Suppose that you have a class function $f(g)$ on G and you know the values $v_1, \ldots, v_n$ on the conjugacy class elements in `conjugacy_classes_representatives(G)` = $[g_1, \ldots, g_n]$. Since the irreducible characters $\rho_1, \ldots, \rho_n$ of G form an E-basis of the space of all class functions (E a "sufficiently large" cyclotomic field), such a class function is a linear combination of these basis elements, $f = c_1\rho_1 + \cdots + c_n\rho_n$. To find the coefficients c_i, you simply solve the linear system `character_table_values(G)` $[v_1, ..., v_n] = [c_1, ..., c_n]$, where $[v_1, \ldots, v_n]$ = `character_table_values(G)` $^{(-1)}[c_1, ..., c_n]$.

AUTHORS:

- David Joyner and William Stein (2006-01-04)

cohomology(*n*, *p=0*)

Compute the group cohomology $H^n(G, F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime.

Wraps HAP's `GroupHomology` function, written by Graham Ellis.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: G.cohomology(1,2)                            # optional - gap_package_
↪hap
Multiplicative Abelian group isomorphic to C2
sage: G = SymmetricGroup(3)
sage: G.cohomology(5)                              # optional - gap_package_
↪hap
Trivial Abelian group
sage: G.cohomology(5,2)                            # optional - gap_package_
↪hap
Multiplicative Abelian group isomorphic to C2
sage: G.homology(5,3)                              # optional - gap_package_
↪hap
Trivial Abelian group
sage: G.homology(5,4)                              # optional - gap_package_
↪hap
Traceback (most recent call last):
...
ValueError: p must be 0 or prime
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
>>> G.cohomology(Integer(1),Integer(2))                        # optional␣
↪- gap_package_hap
Multiplicative Abelian group isomorphic to C2
>>> G = SymmetricGroup(Integer(3))
>>> G.cohomology(Integer(5))                          # optional - gap_
↪package_hap
Trivial Abelian group
>>> G.cohomology(Integer(5),Integer(2))                        # optional␣
↪- gap_package_hap
Multiplicative Abelian group isomorphic to C2
>>> G.homology(Integer(5),Integer(3))                          # optional␣
↪- gap_package_hap
Trivial Abelian group
>>> G.homology(Integer(5),Integer(4))                          # optional␣
↪- gap_package_hap
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: p must be 0 or prime
```

This computes $H^4(S_3, \mathbf{Z})$ and $H^4(S_3, \mathbf{Z}/2\mathbf{Z})$, respectively.

AUTHORS:

- David Joyner and Graham Ellis

REFERENCES:

- G. Ellis, 'Computing group resolutions', J. Symbolic Computation. Vol.38, (2004)1077-1118 (Available at http://hamilton.nuigalway.ie/).
- D. Joyner, 'A primer on computational group homology and cohomology', http://front.math.ucdavis.edu/0706.0549.

cohomology_part(*n*, *p=0*)

Compute the p-part of the group cohomology $H^n(G, F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime.

Wraps HAP's Homology function, written by Graham Ellis, applied to the p-Sylow subgroup of G.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.cohomology_part(7,2)                    # optional - gap_package_hap
Multiplicative Abelian group isomorphic to C2 x C2 x C2
sage: G = SymmetricGroup(3)
sage: G.cohomology_part(2,3)                    # optional - gap_package_hap
Multiplicative Abelian group isomorphic to C3
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> G.cohomology_part(Integer(7),Integer(2))                    # optional -␣
↪gap_package_hap
Multiplicative Abelian group isomorphic to C2 x C2 x C2
>>> G = SymmetricGroup(Integer(3))
>>> G.cohomology_part(Integer(2),Integer(3))                    # optional -␣
↪gap_package_hap
Multiplicative Abelian group isomorphic to C3
```

AUTHORS:

- David Joyner and Graham Ellis

commutator(*other=None*)

Return the commutator subgroup of a group, or of a pair of groups.

INPUT:

- `other` – (default: `None`) a permutation group

OUTPUT:

Let G denote `self`. If `other` is `None`, then this method returns the subgroup of G generated by the set of commutators,

$$\{[g_1, g_2] | g_1, g_2 \in G\} = \{g_1^{-1} g_2^{-1} g_1 g_2 | g_1, g_2 \in G\}$$

Let H denote `other`, in the case that it is not `None`. Then this method returns the group generated by the set of commutators,

$$\{[g, h] | g \in G\, h \in H\} = \{g^{-1} h^{-1} g h | g \in G\, h \in H\}$$

The two groups need only be permutation groups, there is no notion of requiring them to explicitly be subgroups of some other group.

Note: For the identical statement, the generators of the returned group can vary from one execution to the next.

EXAMPLES:

```
sage: G = DiCyclicGroup(4)
sage: G.commutator()
Permutation Group with generators [(1,3,5,7)(2,4,6,8)(9,11,13,15)(10,12,14,
→16)]

sage: G = SymmetricGroup(5)
sage: H = CyclicPermutationGroup(5)
sage: C = G.commutator(H)
sage: C.is_isomorphic(AlternatingGroup(5))
True
```

```
>>> from sage.all import *
>>> G = DiCyclicGroup(Integer(4))
>>> G.commutator()
Permutation Group with generators [(1,3,5,7)(2,4,6,8)(9,11,13,15)(10,12,14,
→16)]

>>> G = SymmetricGroup(Integer(5))
>>> H = CyclicPermutationGroup(Integer(5))
>>> C = G.commutator(H)
>>> C.is_isomorphic(AlternatingGroup(Integer(5)))
True
```

An abelian group will have a trivial commutator.

```
sage: G = CyclicPermutationGroup(10)
sage: G.commutator()
Permutation Group with generators [()]
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(10))
>>> G.commutator()
Permutation Group with generators [()]
```

The quotient of a group by its commutator is always abelian.

```
sage: G = DihedralGroup(20)
sage: C = G.commutator()
sage: Q = G.quotient(C)
sage: Q.is_abelian()
True
```

```
>>> from sage.all import *
>>> G = DihedralGroup(Integer(20))
>>> C = G.commutator()
>>> Q = G.quotient(C)
>>> Q.is_abelian()
True
```

When forming commutators from two groups, the order of the groups does not matter.

```
sage: D = DihedralGroup(3)
sage: S = SymmetricGroup(2)
sage: C1 = D.commutator(S); C1
Permutation Group with generators [(1,2,3)]
sage: C2 = S.commutator(D); C2
Permutation Group with generators [(1,3,2)]
sage: C1 == C2
True
```

```
>>> from sage.all import *
>>> D = DihedralGroup(Integer(3))
>>> S = SymmetricGroup(Integer(2))
>>> C1 = D.commutator(S); C1
Permutation Group with generators [(1,2,3)]
>>> C2 = S.commutator(D); C2
Permutation Group with generators [(1,3,2)]
>>> C1 == C2
True
```

This method calls two different functions in GAP, so this tests that their results are consistent. The commutator groups may have different generators, but the groups are equal.

```
sage: G = DiCyclicGroup(3)
sage: C = G.commutator(); C
Permutation Group with generators [(5,7,6)]
sage: CC = G.commutator(G); CC
Permutation Group with generators [(5,6,7)]
sage: C == CC
True
```

```
>>> from sage.all import *
>>> G = DiCyclicGroup(Integer(3))
>>> C = G.commutator(); C
Permutation Group with generators [(5,7,6)]
>>> CC = G.commutator(G); CC
Permutation Group with generators [(5,6,7)]
>>> C == CC
True
```

The second group is checked.

```
sage: G = SymmetricGroup(2)
sage: G.commutator('junk')
Traceback (most recent call last):
...
TypeError: junk is not a permutation group
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(2))
>>> G.commutator('junk')
Traceback (most recent call last):
...
TypeError: junk is not a permutation group
```

composition_series()

Return the composition series of this group as a list of permutation groups.

EXAMPLES:

These computations use pseudo-random numbers, so we set the seed for reproducible testing.

```
sage: set_random_seed(0)
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.composition_series()
[Subgroup generated by [(3,4), (1,2,3)(4,5)] of
  (Permutation Group with generators [(3,4), (1,2,3)(4,5)]),
 Subgroup generated by [(1,5)(3,4), (1,5)(2,4), (1,3,5)] of
  (Permutation Group with generators [(3,4), (1,2,3)(4,5)]),
 Subgroup generated by [()] of
  (Permutation Group with generators [(3,4), (1,2,3)(4,5)])]
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: CS = G.composition_series()
sage: CS[3]
Subgroup generated by [()] of
 (Permutation Group with generators [(1,2), (1,2,3)(4,5)])
```

```
>>> from sage.all import *
>>> set_random_seed(Integer(0))
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))],[(Integer(3),Integer(4))]])
>>> G.composition_series()
[Subgroup generated by [(3,4), (1,2,3)(4,5)] of
  (Permutation Group with generators [(3,4), (1,2,3)(4,5)]),
 Subgroup generated by [(1,5)(3,4), (1,5)(2,4), (1,3,5)] of
  (Permutation Group with generators [(3,4), (1,2,3)(4,5)]),
 Subgroup generated by [()] of
  (Permutation Group with generators [(3,4), (1,2,3)(4,5)])]
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))], [(Integer(1),Integer(2))]])
>>> CS = G.composition_series()
>>> CS[Integer(3)]
Subgroup generated by [()] of
 (Permutation Group with generators [(1,2), (1,2,3)(4,5)])
```

conjugacy_class(*g*)

Return the conjugacy class of `g` inside the group `self`.

INPUT:

- `g` – an element of the permutation group `self`

OUTPUT:

The conjugacy class of `g` in the group `self`. If `self` is the group denoted by G, this method computes the set $\{x^{-1}gx \mid x \in G\}$

EXAMPLES:

```
sage: G = DihedralGroup(3)
sage: g = G.gen(0)
sage: G.conjugacy_class(g)
Conjugacy class of (1,2,3) in Dihedral group of order 6 as a permutation group
```

```
>>> from sage.all import *
>>> G = DihedralGroup(Integer(3))
>>> g = G.gen(Integer(0))
>>> G.conjugacy_class(g)
Conjugacy class of (1,2,3) in Dihedral group of order 6 as a permutation group
```

conjugacy_classes()

Return a list with all the conjugacy classes of `self`.

EXAMPLES:

```
sage: G = DihedralGroup(3)
sage: G.conjugacy_classes()
[Conjugacy class of () in Dihedral group of order 6 as a permutation group,
 Conjugacy class of (2,3) in Dihedral group of order 6 as a permutation group,
 Conjugacy class of (1,2,3) in Dihedral group of order 6 as a permutation␣
↪group]
```

```
>>> from sage.all import *
>>> G = DihedralGroup(Integer(3))
>>> G.conjugacy_classes()
[Conjugacy class of () in Dihedral group of order 6 as a permutation group,
 Conjugacy class of (2,3) in Dihedral group of order 6 as a permutation group,
 Conjugacy class of (1,2,3) in Dihedral group of order 6 as a permutation␣
↪group]
```

conjugacy_classes_representatives()

Return a complete list of representatives of conjugacy classes in a permutation group G.

The ordering is that given by GAP.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: cl = G.conjugacy_classes_representatives(); cl
[(), (2,4), (1,2)(3,4), (1,2,3,4), (1,3)(2,4)]
sage: cl[3] in G
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],␣
↪[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> cl = G.conjugacy_classes_representatives(); cl
[(), (2,4), (1,2)(3,4), (1,2,3,4), (1,3)(2,4)]
>>> cl[Integer(3)] in G
True
```

```
sage: G = SymmetricGroup(5)
sage: G.conjugacy_classes_representatives()                              #␣
↪needs sage.combinat
[(), (1,2), (1,2)(3,4), (1,2,3), (1,2,3)(4,5), (1,2,3,4), (1,2,3,4,5)]
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> G.conjugacy_classes_representatives()                                #␣
↪needs sage.combinat
[(), (1,2), (1,2)(3,4), (1,2,3), (1,2,3)(4,5), (1,2,3,4), (1,2,3,4,5)]
```

```
sage: S = SymmetricGroup(['a','b','c'])
sage: S.conjugacy_classes_representatives()                              #␣
↪needs sage.combinat
[(), ('a','b'), ('a','b','c')]
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(['a','b','c'])
>>> S.conjugacy_classes_representatives()                                #␣
↪needs sage.combinat
[(), ('a','b'), ('a','b','c')]
```

AUTHORS:

- David Joyner and William Stein (2006-01-04)

conjugacy_classes_subgroups()

Return a complete list of representatives of conjugacy classes of subgroups in a permutation group G.

The ordering is that given by GAP.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: cl = G.conjugacy_classes_subgroups(); cl
[Subgroup generated by [()] of
  (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]),
 Subgroup generated by [(1,2)(3,4)] of
  (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]),
 Subgroup generated by [(1,3)(2,4)] of
  (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]),
 Subgroup generated by [(2,4)] of
  (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]),
 Subgroup generated by [(1,4)(2,3), (1,2)(3,4)] of
  (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]),
```

(continues on next page)

(continued from previous page)

```
 Subgroup generated by [(2,4), (1,3)(2,4)] of
   (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]),
 Subgroup generated by [(1,2,3,4), (1,3)(2,4)] of
   (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]),
 Subgroup generated by [(1,4)(2,3), (1,2)(3,4), (2,4)] of
   (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],␣
↪[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> cl = G.conjugacy_classes_subgroups(); cl
[Subgroup generated by [()] of
   (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]),
 Subgroup generated by [(1,2)(3,4)] of
   (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]),
 Subgroup generated by [(1,3)(2,4)] of
   (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]),
 Subgroup generated by [(2,4)] of
   (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]),
 Subgroup generated by [(1,4)(2,3), (1,2)(3,4)] of
   (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]),
 Subgroup generated by [(2,4), (1,3)(2,4)] of
   (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]),
 Subgroup generated by [(1,2,3,4), (1,3)(2,4)] of
   (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]),
 Subgroup generated by [(1,4)(2,3), (1,2)(3,4), (2,4)] of
   (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])]
```

```
sage: G = SymmetricGroup(3)
sage: G.conjugacy_classes_subgroups()
[Subgroup generated by [()] of
   (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(2,3)] of
   (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(1,2,3)] of
   (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(1,2,3), (2,3)] of
   (Symmetric group of order 3! as a permutation group)]
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> G.conjugacy_classes_subgroups()
[Subgroup generated by [()] of
   (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(2,3)] of
   (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(1,2,3)] of
   (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(1,2,3), (2,3)] of
   (Symmetric group of order 3! as a permutation group)]
```

AUTHORS:

- David Joyner (2006-10)

conjugate(*g*)

Return the group formed by conjugating `self` with `g`.

INPUT:

- `g` – a permutation group element, or an object that converts to a permutation group element, such as a list of integers or a string of cycles.

OUTPUT:

If `self` is the group denoted by H, then this method computes the group

$$g^{-1}Hg = \{g^{-1}hg|h \in H\}$$

which is the group H conjugated by g.

There are no restrictions on `self` and `g` belonging to a common permutation group, and correspondingly, there is no relationship (such as a common parent) between `self` and the output group.

EXAMPLES:

```
sage: G = DihedralGroup(6)
sage: a = PermutationGroupElement("(1,2,3,4)")
sage: G.conjugate(a)
Permutation Group with generators [(1,5,6,2,3,4), (1,4)(2,6)(3,5)]
```

```
>>> from sage.all import *
>>> G = DihedralGroup(Integer(6))
>>> a = PermutationGroupElement("(1,2,3,4)")
>>> G.conjugate(a)
Permutation Group with generators [(1,5,6,2,3,4), (1,4)(2,6)(3,5)]
```

The element performing the conjugation can be specified in several ways.

```
sage: G = DihedralGroup(6)
sage: strng = "(1,2,3,4)"
sage: G.conjugate(strng)
Permutation Group with generators [(1,5,6,2,3,4), (1,4)(2,6)(3,5)]
sage: G = DihedralGroup(6)
sage: lst = [2,3,4,1]
sage: G.conjugate(lst)
Permutation Group with generators [(1,5,6,2,3,4), (1,4)(2,6)(3,5)]
sage: G = DihedralGroup(6)
sage: cycles = [(1,2,3,4)]
sage: G.conjugate(cycles)
Permutation Group with generators [(1,5,6,2,3,4), (1,4)(2,6)(3,5)]
```

```
>>> from sage.all import *
>>> G = DihedralGroup(Integer(6))
>>> strng = "(1,2,3,4)"
>>> G.conjugate(strng)
Permutation Group with generators [(1,5,6,2,3,4), (1,4)(2,6)(3,5)]
>>> G = DihedralGroup(Integer(6))
>>> lst = [Integer(2),Integer(3),Integer(4),Integer(1)]
```

(continues on next page)

(continued from previous page)

```
>>> G.conjugate(lst)
Permutation Group with generators [(1,5,6,2,3,4), (1,4)(2,6)(3,5)]
>>> G = DihedralGroup(Integer(6))
>>> cycles = [(Integer(1),Integer(2),Integer(3),Integer(4))]
>>> G.conjugate(cycles)
Permutation Group with generators [(1,5,6,2,3,4), (1,4)(2,6)(3,5)]
```

Conjugation is a group automorphism, so conjugate groups will be isomorphic.

```
sage: G = DiCyclicGroup(6)
sage: G.degree()
11
sage: cycle = [i+1 for i in range(1,11)] + [1]
sage: C = G.conjugate(cycle)
sage: G.is_isomorphic(C)
True
```

```
>>> from sage.all import *
>>> G = DiCyclicGroup(Integer(6))
>>> G.degree()
11
>>> cycle = [i+Integer(1) for i in range(Integer(1),Integer(11))] +␣
↪[Integer(1)]
>>> C = G.conjugate(cycle)
>>> G.is_isomorphic(C)
True
```

The conjugating element may be from a symmetric group with larger degree than the group being conjugated.

```
sage: G = AlternatingGroup(5)
sage: G.degree()
5
sage: g = "(1,3)(5,6,7)"
sage: H = G.conjugate(g); H
Permutation Group with generators [(1,4,6,3,2), (1,4,6)]
sage: H.degree()
6
```

```
>>> from sage.all import *
>>> G = AlternatingGroup(Integer(5))
>>> G.degree()
5
>>> g = "(1,3)(5,6,7)"
>>> H = G.conjugate(g); H
Permutation Group with generators [(1,4,6,3,2), (1,4,6)]
>>> H.degree()
6
```

The conjugating element is checked.

```
sage: G = SymmetricGroup(3)
sage: G.conjugate("junk")
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: junk does not convert to a permutation group element
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> G.conjugate("junk")
Traceback (most recent call last):
...
TypeError: junk does not convert to a permutation group element
```

construction()

Return the construction of `self`.

EXAMPLES:

```
sage: P1 = PermutationGroup([[(1,2)]])
sage: P1.construction()
(PermutationGroupFunctor[(1,2)], Permutation Group with generators [()])

sage: PermutationGroup([]).construction() is None
True
```

```
>>> from sage.all import *
>>> P1 = PermutationGroup([[(Integer(1),Integer(2))]])
>>> P1.construction()
(PermutationGroupFunctor[(1,2)], Permutation Group with generators [()])

>>> PermutationGroup([]).construction() is None
True
```

This allows us to perform computations like the following:

```
sage: P1 = PermutationGroup([[(1,2)]]); p1 = P1.gen()
sage: P2 = PermutationGroup([[(1,3)]]); p2 = P2.gen()
sage: p = p1*p2; p
(1,2,3)
sage: p.parent()
Permutation Group with generators [(1,2), (1,3)]
sage: p.parent().domain()
{1, 2, 3}
```

```
>>> from sage.all import *
>>> P1 = PermutationGroup([[(Integer(1),Integer(2))]]); p1 = P1.gen()
>>> P2 = PermutationGroup([[(Integer(1),Integer(3))]]); p2 = P2.gen()
>>> p = p1*p2; p
(1,2,3)
>>> p.parent()
Permutation Group with generators [(1,2), (1,3)]
>>> p.parent().domain()
{1, 2, 3}
```

Note that this will merge permutation groups with different domains:

```
sage: g1 = PermutationGroupElement([(1,2),(3,4,5)])
sage: g2 = PermutationGroup([('a','b')], domain=['a', 'b']).gens()[0]
sage: g2
('a','b')
sage: p = g1*g2; p
(1,2)(3,4,5)('a','b')
sage: P = parent(p)
sage: P
Permutation Group with generators [('a','b'), (1,2), (1,2,3,4,5)]
```

```
>>> from sage.all import *
>>> g1 = PermutationGroupElement([(Integer(1),Integer(2)),(Integer(3),
↪Integer(4),Integer(5))])
>>> g2 = PermutationGroup([('a','b')], domain=['a', 'b']).gens()[Integer(0)]
>>> g2
('a','b')
>>> p = g1*g2; p
(1,2)(3,4,5)('a','b')
>>> P = parent(p)
>>> P
Permutation Group with generators [('a','b'), (1,2), (1,2,3,4,5)]
```

cosets(*S*, *side='right'*)

Return a list of the cosets of `S` in `self`.

INPUT:

- `S` – a subgroup of `self`; an error is raised if `S` is not a subgroup
- `side` – (default: `'right'`) determines if right cosets or left cosets are returned. `side` refers to where the representative is placed in the products forming the cosets and thus allowable values are only `'right'` and `'left'`.

OUTPUT:

A list of lists. Each inner list is a coset of the subgroup in the group. The first element of each coset is the smallest element (based on the ordering of the elements of `self`) of all the group elements that have not yet appeared in a previous coset. The elements of each coset are in the same order as the subgroup elements used to build the coset's elements.

As a consequence, the subgroup itself is the first coset, and its first element is the identity element. For each coset, the first element listed is the element used as a representative to build the coset. These representatives form an increasing sequence across the list of cosets, and within a coset the representative is the smallest element of its coset (both orderings are based on of the ordering of elements of `self`).

In the case of a normal subgroup, left and right cosets should appear in the same order as part of the outer list. However, the list of the elements of a particular coset may be in a different order for the right coset versus the order in the left coset. So, if you check to see if a subgroup is normal, it is necessary to sort each individual coset first (but not the list of cosets, due to the ordering of the representatives). See below for examples of this.

Note

This is a naive implementation intended for instructional purposes, and hence is slow for larger groups. Sage and GAP provide more sophisticated functions for working quickly with cosets of larger groups.

EXAMPLES:

The default is to build right cosets. This example works with the symmetry group of an 8-gon and a normal subgroup. Notice that a straight check on the equality of the output is not sufficient to check normality, while sorting the individual cosets is sufficient to then simply test equality of the list of lists. Study the second coset in each list to understand the need for sorting the elements of the cosets.

```
sage: G = DihedralGroup(8)
sage: quarter_turn = G('(1,3,5,7)(2,4,6,8)'); quarter_turn
(1,3,5,7)(2,4,6,8)
sage: S = G.subgroup([quarter_turn])
sage: rc = G.cosets(S); rc
[[(), (1,3,5,7)(2,4,6,8), (1,5)(2,6)(3,7)(4,8), (1,7,5,3)(2,8,6,4)],
 [(2,8)(3,7)(4,6), (1,7)(2,6)(3,5), (1,5)(2,4)(6,8), (1,3)(4,8)(5,7)],
 [(1,2)(3,8)(4,7)(5,6), (1,8)(2,7)(3,6)(4,5), (1,6)(2,5)(3,4)(7,8), (1,4)(2,
↪3)(5,8)(6,7)],
 [(1,2,3,4,5,6,7,8), (1,4,7,2,5,8,3,6), (1,6,3,8,5,2,7,4), (1,8,7,6,5,4,3,2)]]
sage: lc = G.cosets(S, side='left'); lc
[[(), (1,3,5,7)(2,4,6,8), (1,5)(2,6)(3,7)(4,8), (1,7,5,3)(2,8,6,4)],
 [(2,8)(3,7)(4,6), (1,3)(4,8)(5,7), (1,5)(2,4)(6,8), (1,7)(2,6)(3,5)],
 [(1,2)(3,8)(4,7)(5,6), (1,4)(2,3)(5,8)(6,7), (1,6)(2,5)(3,4)(7,8), (1,8)(2,
↪7)(3,6)(4,5)],
 [(1,2,3,4,5,6,7,8), (1,4,7,2,5,8,3,6), (1,6,3,8,5,2,7,4), (1,8,7,6,5,4,3,2)]]

sage: S.is_normal(G)
True
sage: rc == lc
False
sage: rc_sorted = [sorted(c) for c in rc]
sage: lc_sorted = [sorted(c) for c in lc]
sage: rc_sorted == lc_sorted
True
```

```
>>> from sage.all import *
>>> G = DihedralGroup(Integer(8))
>>> quarter_turn = G('(1,3,5,7)(2,4,6,8)'); quarter_turn
(1,3,5,7)(2,4,6,8)
>>> S = G.subgroup([quarter_turn])
>>> rc = G.cosets(S); rc
[[(), (1,3,5,7)(2,4,6,8), (1,5)(2,6)(3,7)(4,8), (1,7,5,3)(2,8,6,4)],
 [(2,8)(3,7)(4,6), (1,7)(2,6)(3,5), (1,5)(2,4)(6,8), (1,3)(4,8)(5,7)],
 [(1,2)(3,8)(4,7)(5,6), (1,8)(2,7)(3,6)(4,5), (1,6)(2,5)(3,4)(7,8), (1,4)(2,
↪3)(5,8)(6,7)],
 [(1,2,3,4,5,6,7,8), (1,4,7,2,5,8,3,6), (1,6,3,8,5,2,7,4), (1,8,7,6,5,4,3,2)]]
>>> lc = G.cosets(S, side='left'); lc
[[(), (1,3,5,7)(2,4,6,8), (1,5)(2,6)(3,7)(4,8), (1,7,5,3)(2,8,6,4)],
 [(2,8)(3,7)(4,6), (1,3)(4,8)(5,7), (1,5)(2,4)(6,8), (1,7)(2,6)(3,5)],
 [(1,2)(3,8)(4,7)(5,6), (1,4)(2,3)(5,8)(6,7), (1,6)(2,5)(3,4)(7,8), (1,8)(2,
↪7)(3,6)(4,5)],
 [(1,2,3,4,5,6,7,8), (1,4,7,2,5,8,3,6), (1,6,3,8,5,2,7,4), (1,8,7,6,5,4,3,2)]]

>>> S.is_normal(G)
True
>>> rc == lc
```

(continues on next page)

(continued from previous page)

```
False
>>> rc_sorted = [sorted(c) for c in rc]
>>> lc_sorted = [sorted(c) for c in lc]
>>> rc_sorted == lc_sorted
True
```

An example with the symmetry group of a regular tetrahedron and a subgroup that is not normal. Thus, the right and left cosets are different (and so are the representatives). With each individual coset sorted, a naive test of normality is possible.

```
sage: A = AlternatingGroup(4)
sage: face_turn = A('(1,2,3)'); face_turn
(1,2,3)
sage: stabilizer = A.subgroup([face_turn])
sage: rc = A.cosets(stabilizer, side='right'); rc
[[(), (1,2,3), (1,3,2)],
 [(2,3,4), (1,3)(2,4), (1,4,2)],
 [(2,4,3), (1,4,3), (1,2)(3,4)],
 [(1,2,4), (1,4)(2,3), (1,3,4)]]
sage: lc = A.cosets(stabilizer, side='left'); lc
[[(), (1,2,3), (1,3,2)],
 [(2,3,4), (1,2)(3,4), (1,3,4)],
 [(2,4,3), (1,2,4), (1,3)(2,4)],
 [(1,4,2), (1,4,3), (1,4)(2,3)]]

sage: stabilizer.is_normal(A)
False
sage: rc_sorted = [sorted(c) for c in rc]
sage: lc_sorted = [sorted(c) for c in lc]
sage: rc_sorted == lc_sorted
False
```

```
>>> from sage.all import *
>>> A = AlternatingGroup(Integer(4))
>>> face_turn = A('(1,2,3)'); face_turn
(1,2,3)
>>> stabilizer = A.subgroup([face_turn])
>>> rc = A.cosets(stabilizer, side='right'); rc
[[(), (1,2,3), (1,3,2)],
 [(2,3,4), (1,3)(2,4), (1,4,2)],
 [(2,4,3), (1,4,3), (1,2)(3,4)],
 [(1,2,4), (1,4)(2,3), (1,3,4)]]
>>> lc = A.cosets(stabilizer, side='left'); lc
[[(), (1,2,3), (1,3,2)],
 [(2,3,4), (1,2)(3,4), (1,3,4)],
 [(2,4,3), (1,2,4), (1,3)(2,4)],
 [(1,4,2), (1,4,3), (1,4)(2,3)]]

>>> stabilizer.is_normal(A)
False
>>> rc_sorted = [sorted(c) for c in rc]
>>> lc_sorted = [sorted(c) for c in lc]
```

(continues on next page)

(continued from previous page)

```
>>> rc_sorted == lc_sorted
False
```

AUTHOR:

- Rob Beezer (2011-01-31)

degree()

Return the degree of this permutation group.

EXAMPLES:

```
sage: S = SymmetricGroup(['a','b','c'])
sage: S.degree()
3
sage: G = PermutationGroup([(1,3),(4,5)])
sage: G.degree()
5
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(['a','b','c'])
>>> S.degree()
3
>>> G = PermutationGroup([(Integer(1),Integer(3)),(Integer(4),Integer(5))])
>>> G.degree()
5
```

Note that you can explicitly specify the domain to get a permutation group of smaller degree:

```
sage: G = PermutationGroup([(1,3),(4,5)], domain=[1,3,4,5])
sage: G.degree()
4
```

```
>>> from sage.all import *
>>> G = PermutationGroup([(Integer(1),Integer(3)),(Integer(4),Integer(5))],␣
↪domain=[Integer(1),Integer(3),Integer(4),Integer(5)])
>>> G.degree()
4
```

derived_series()

Return the derived series of this group as a list of permutation groups.

EXAMPLES:

These computations use pseudo-random numbers, so we set the seed for reproducible testing.

```
sage: set_random_seed(0)
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.derived_series()
[Subgroup generated by [(3,4), (1,2,3)(4,5)] of
  (Permutation Group with generators [(3,4), (1,2,3)(4,5)]),
 Subgroup generated by [(1,5)(3,4), (1,5)(2,4), (1,3,5)] of
  (Permutation Group with generators [(3,4), (1,2,3)(4,5)])]
```

```
>>> from sage.all import *
>>> set_random_seed(Integer(0))
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))],[(Integer(3),Integer(4))]])
>>> G.derived_series()
[Subgroup generated by [(3,4), (1,2,3)(4,5)] of
  (Permutation Group with generators [(3,4), (1,2,3)(4,5)]),
 Subgroup generated by [(1,5)(3,4), (1,5)(2,4), (1,3,5)] of
  (Permutation Group with generators [(3,4), (1,2,3)(4,5)])]
```

direct_product(*other*, *maps=True*)

Wraps GAP's `DirectProduct`, `Embedding`, and `Projection`.

Sage calls GAP's `DirectProduct`, which chooses an efficient representation for the direct product. The direct product of permutation groups will be a permutation group again. For a direct product `D`, the GAP operation `Embedding(D,i)` returns the homomorphism embedding the i-th factor into `D`. The GAP operation `Projection(D,i)` gives the projection of `D` onto the i-th factor. This method returns a 5-tuple: a permutation group and 4 morphisms.

INPUT:

- `self`, `other` – permutation groups

OUTPUT:

- `D` – a direct product of the inputs, returned as a permutation group as well
- `iota1` – an embedding of `self` into `D`
- `iota2` – an embedding of `other` into `D`
- `pr1` – the projection of `D` onto `self` (giving a splitting `1 - other - D - self - 1`)
- `pr2` – the projection of `D` onto `other` (giving a splitting `1 - self - D - other - 1`)

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: D = G.direct_product(G, False); D
Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
sage: D,iota1,iota2,pr1,pr2 = G.direct_product(G)
sage: D; iota1; iota2; pr1; pr2
Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
Permutation group morphism:
  From: Cyclic group of order 4 as a permutation group
  To:   Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
  Defn: Embedding( Group( [ (1,2,3,4), (5,6,7,8) ] ), 1 )
Permutation group morphism:
  From: Cyclic group of order 4 as a permutation group
  To:   Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
  Defn: Embedding( Group( [ (1,2,3,4), (5,6,7,8) ] ), 2 )
Permutation group morphism:
  From: Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
  To:   Cyclic group of order 4 as a permutation group
  Defn: Projection( Group( [ (1,2,3,4), (5,6,7,8) ] ), 1 )
Permutation group morphism:
  From: Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
  To:   Cyclic group of order 4 as a permutation group
```

(continues on next page)

(continued from previous page)

```
  Defn: Projection( Group( [ (1,2,3,4), (5,6,7,8) ] ), 2 )
sage: g = D([(1,3),(2,4)]); g
(1,3)(2,4)
sage: d = D([(1,4,3,2),(5,7),(6,8)]); d
(1,4,3,2)(5,7)(6,8)
sage: iota1(g); iota2(g); pr1(d); pr2(d)
(1,3)(2,4)
(5,7)(6,8)
(1,4,3,2)
(1,3)(2,4)
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(4))
>>> D = G.direct_product(G, False); D
Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
>>> D,iota1,iota2,pr1,pr2 = G.direct_product(G)
>>> D; iota1; iota2; pr1; pr2
Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
Permutation group morphism:
  From: Cyclic group of order 4 as a permutation group
  To:   Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
  Defn: Embedding( Group( [ (1,2,3,4), (5,6,7,8) ] ), 1 )
Permutation group morphism:
  From: Cyclic group of order 4 as a permutation group
  To:   Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
  Defn: Embedding( Group( [ (1,2,3,4), (5,6,7,8) ] ), 2 )
Permutation group morphism:
  From: Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
  To:   Cyclic group of order 4 as a permutation group
  Defn: Projection( Group( [ (1,2,3,4), (5,6,7,8) ] ), 1 )
Permutation group morphism:
  From: Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
  To:   Cyclic group of order 4 as a permutation group
  Defn: Projection( Group( [ (1,2,3,4), (5,6,7,8) ] ), 2 )
>>> g = D([(Integer(1),Integer(3)),(Integer(2),Integer(4))]); g
(1,3)(2,4)
>>> d = D([(Integer(1),Integer(4),Integer(3),Integer(2)),(Integer(5),
↪Integer(7)),(Integer(6),Integer(8))]); d
(1,4,3,2)(5,7)(6,8)
>>> iota1(g); iota2(g); pr1(d); pr2(d)
(1,3)(2,4)
(5,7)(6,8)
(1,4,3,2)
(1,3)(2,4)
```

disjoint_direct_product_decomposition()

Return the finest partition of the underlying set such that `self` is isomorphic to the direct product of the projections of `self` onto each part of the partition. Each part is a union of orbits of `self`.

The algorithm is from [CJ2022], which runs in time polynomial in $n \cdot |X|$, where n is the degree of the group and $|X|$ is the size of a generating set, see Theorem 4.5.

EXAMPLES:

The example from the original paper:

```
sage: H = PermutationGroup([[(1,2,3),(7,9,8),(10,12,11)],[(4,5,6),(7,8,9),(10,
↪11,12)],[(5,6),(8,9),(11,12)],[(7,8,9),(10,11,12)]])
sage: S = H.disjoint_direct_product_decomposition(); S
{{1, 2, 3}, {4, 5, 6, 7, 8, 9, 10, 11, 12}}
sage: A = libgap.Stabilizer(H, list(S[0]), libgap.OnTuples); A
Group([ (7,8,9)(10,11,12), (5,6)(8,9)(11,12), (4,5,6)(7,8,9)(10,11,12) ])
sage: B = libgap.Stabilizer(H, list(S[1]), libgap.OnTuples); B
Group([ (1,2,3) ])
sage: T = PermutationGroup(gap_group=libgap.DirectProduct(A,B))
sage: T.is_isomorphic(H)
True
sage: PermutationGroup(PermutationGroup(gap_group=A).gens(),
↪domain=list(S[1])).disjoint_direct_product_decomposition()
{{4, 5, 6, 7, 8, 9, 10, 11, 12}}
sage: PermutationGroup(PermutationGroup(gap_group=B).gens(),
↪domain=list(S[0])).disjoint_direct_product_decomposition()
{{1, 2, 3}}
```

```
>>> from sage.all import *
>>> H = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(7),
↪Integer(9),Integer(8)),(Integer(10),Integer(12),Integer(11))],[(Integer(4),
↪Integer(5),Integer(6)),(Integer(7),Integer(8),Integer(9)),(Integer(10),
↪Integer(11),Integer(12))],[(Integer(5),Integer(6)),(Integer(8),Integer(9)),
↪(Integer(11),Integer(12))],[(Integer(7),Integer(8),Integer(9)),(Integer(10),
↪Integer(11),Integer(12))]])
>>> S = H.disjoint_direct_product_decomposition(); S
{{1, 2, 3}, {4, 5, 6, 7, 8, 9, 10, 11, 12}}
>>> A = libgap.Stabilizer(H, list(S[Integer(0)]), libgap.OnTuples); A
Group([ (7,8,9)(10,11,12), (5,6)(8,9)(11,12), (4,5,6)(7,8,9)(10,11,12) ])
>>> B = libgap.Stabilizer(H, list(S[Integer(1)]), libgap.OnTuples); B
Group([ (1,2,3) ])
>>> T = PermutationGroup(gap_group=libgap.DirectProduct(A,B))
>>> T.is_isomorphic(H)
True
>>> PermutationGroup(PermutationGroup(gap_group=A).gens(),
↪domain=list(S[Integer(1)])).disjoint_direct_product_decomposition()
{{4, 5, 6, 7, 8, 9, 10, 11, 12}}
>>> PermutationGroup(PermutationGroup(gap_group=B).gens(),
↪domain=list(S[Integer(0)])).disjoint_direct_product_decomposition()
{{1, 2, 3}}
```

An example with a different domain:

```
sage: PermutationGroup([[('a','c','d'),('b','e')]]).disjoint_direct_product_
↪decomposition()
{{'a', 'c', 'd'}, {'b', 'e'}}
sage: PermutationGroup([[('a','c','d','b','e')]]).disjoint_direct_product_
↪decomposition()
{{'a', 'b', 'c', 'd', 'e'}}
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> PermutationGroup([[('a','c','d'),('b','e')]]).disjoint_direct_product_
↪decomposition()
{{'a', 'c', 'd'}, {'b', 'e'}}
>>> PermutationGroup([[('a','c','d','b','e')]]).disjoint_direct_product_
↪decomposition()
{{'a', 'b', 'c', 'd', 'e'}}
```

Counting the number of "connected" permutation groups of degree n:

```
sage: seq = [sum(1 for G in SymmetricGroup(n).conjugacy_classes_subgroups()␣
↪if len(G.disjoint_direct_product_decomposition()) == 1) for n in range(1,
↪8)]; seq
[1, 1, 2, 6, 6, 27, 20]
sage: oeis(seq) # optional -- internet
0: A005226: Number of atomic species of degree n; also number of connected␣
↪permutation groups of degree n.
```

```
>>> from sage.all import *
>>> seq = [sum(Integer(1) for G in SymmetricGroup(n).conjugacy_classes_
↪subgroups() if len(G.disjoint_direct_product_decomposition()) ==␣
↪Integer(1)) for n in range(Integer(1),Integer(8))]; seq
[1, 1, 2, 6, 6, 27, 20]
>>> oeis(seq) # optional -- internet
0: A005226: Number of atomic species of degree n; also number of connected␣
↪permutation groups of degree n.
```

domain()

Return the underlying set that this permutation group acts on.

EXAMPLES:

```
sage: P = PermutationGroup([(1,2),(3,5)])
sage: P.domain()
{1, 2, 3, 4, 5}
sage: S = SymmetricGroup(['a', 'b', 'c'])
sage: S.domain()
{'a', 'b', 'c'}
```

```
>>> from sage.all import *
>>> P = PermutationGroup([(Integer(1),Integer(2)),(Integer(3),Integer(5))])
>>> P.domain()
{1, 2, 3, 4, 5}
>>> S = SymmetricGroup(['a', 'b', 'c'])
>>> S.domain()
{'a', 'b', 'c'}
```

exponent()

Compute the exponent of the group.

The exponent e of a group G is the LCM of the orders of its elements, that is, e is the smallest integer such that $g^e = 1$ for all $g \in G$.

EXAMPLES:

```
sage: G = AlternatingGroup(4)
sage: G.exponent()
6
```

```
>>> from sage.all import *
>>> G = AlternatingGroup(Integer(4))
>>> G.exponent()
6
```

fitting_subgroup()

Return the Fitting subgroup of `self`.

The Fitting subgroup of a group G is the largest nilpotent normal subgroup of G.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3,4)],[(2,4)]])
sage: G.fitting_subgroup()
Subgroup generated by [(2,4), (1,2,3,4), (1,3)] of
 (Permutation Group with generators [(2,4), (1,2,3,4)])
sage: G = PermutationGroup([[(1,2,3,4)],[(1,2)]])
sage: G.fitting_subgroup()
Subgroup generated by [(1,2)(3,4), (1,3)(2,4)] of
 (Permutation Group with generators [(1,2), (1,2,3,4)])
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))],
↪[(Integer(2),Integer(4))]])
>>> G.fitting_subgroup()
Subgroup generated by [(2,4), (1,2,3,4), (1,3)] of
 (Permutation Group with generators [(2,4), (1,2,3,4)])
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))],
↪[(Integer(1),Integer(2))]])
>>> G.fitting_subgroup()
Subgroup generated by [(1,2)(3,4), (1,3)(2,4)] of
 (Permutation Group with generators [(1,2), (1,2,3,4)])
```

fixed_points()

Return the list of points fixed by `self`, i.e., the subset of `.domain()` not moved by any element of `self`.

EXAMPLES:

```
sage: G = PermutationGroup([(1,2,3)])
sage: G.fixed_points()
[]
sage: G = PermutationGroup([(1,2,3),(5,6)])
sage: G.fixed_points()
[4]
sage: G = PermutationGroup([[(1,4,7)],[(4,3),(6,7)]])
sage: G.fixed_points()
[2, 5]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([(Integer(1),Integer(2),Integer(3))])
>>> G.fixed_points()
[]
>>> G = PermutationGroup([(Integer(1),Integer(2),Integer(3)),(Integer(5),
↪Integer(6))])
>>> G.fixed_points()
[4]
>>> G = PermutationGroup([[(Integer(1),Integer(4),Integer(7))],[(Integer(4),
↪Integer(3)),(Integer(6),Integer(7))]])
>>> G.fixed_points()
[2, 5]
```

frattini_subgroup()

Return the Frattini subgroup of `self`.

The Frattini subgroup of a group G is the intersection of all maximal subgroups of G.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3,4)],[(2,4)]])
sage: G.frattini_subgroup()
Subgroup generated by [(1,3)(2,4)] of
 (Permutation Group with generators [(2,4), (1,2,3,4)])
sage: G = SymmetricGroup(4)
sage: G.frattini_subgroup()
Subgroup generated by [()] of
 (Symmetric group of order 4! as a permutation group)
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))],
↪[(Integer(2),Integer(4))]])
>>> G.frattini_subgroup()
Subgroup generated by [(1,3)(2,4)] of
 (Permutation Group with generators [(2,4), (1,2,3,4)])
>>> G = SymmetricGroup(Integer(4))
>>> G.frattini_subgroup()
Subgroup generated by [()] of
 (Symmetric group of order 4! as a permutation group)
```

gap()

This method from *sage.groups.libgap_wrapper.ParentLibGAP* is added in order to achieve compatibility and have *sage.groups.libgap_morphism.GroupHomset_libgap* work for permutation groups, as well

OUTPUT: an instance of `sage.libs.gap.element.GapElement` representing this group

EXAMPLES:

```
sage: P8 = PSp(8,3)
sage: P8.gap()
<permutation group of size 65784756654489600 with 2 generators>
sage: gap(P8) == P8.gap()
False
```

(continues on next page)

(continued from previous page)

```
sage: S3 = SymmetricGroup(3)
sage: S3.gap()
Sym( [ 1 .. 3 ] )
sage: gap(S3) == S3.gap()
False
```

```
>>> from sage.all import *
>>> P8 = PSp(Integer(8),Integer(3))
>>> P8.gap()
<permutation group of size 65784756654489600 with 2 generators>
>>> gap(P8) == P8.gap()
False
>>> S3 = SymmetricGroup(Integer(3))
>>> S3.gap()
Sym( [ 1 .. 3 ] )
>>> gap(S3) == S3.gap()
False
```

gen(*i=None*)

Return the i-th generator of `self`; that is, the i-th element of the list `self.gens()`.

The argument i may be omitted if there is only one generator (but this will raise an error otherwise).

EXAMPLES:

We explicitly construct the alternating group on four elements:

```
sage: A4 = PermutationGroup([[(1,2,3)],[(2,3,4)]]); A4
Permutation Group with generators [(2,3,4), (1,2,3)]
sage: A4.gens()
((2,3,4), (1,2,3))
sage: A4.gen(0)
(2,3,4)
sage: A4.gen(1)
(1,2,3)
sage: A4.gens()[0]; A4.gens()[1]
(2,3,4)
(1,2,3)

sage: P1 = PermutationGroup([[(1,2)]]); P1.gen()
(1,2)
```

```
>>> from sage.all import *
>>> A4 = PermutationGroup([[(Integer(1),Integer(2),Integer(3))],[(Integer(2),
↪Integer(3),Integer(4))]]); A4
Permutation Group with generators [(2,3,4), (1,2,3)]
>>> A4.gens()
((2,3,4), (1,2,3))
>>> A4.gen(Integer(0))
(2,3,4)
>>> A4.gen(Integer(1))
(1,2,3)
>>> A4.gens()[Integer(0)]; A4.gens()[Integer(1)]
```

(continues on next page)

(continued from previous page)

```
(2,3,4)
(1,2,3)

>>> P1 = PermutationGroup([[(Integer(1),Integer(2))]]); P1.gen()
(1,2)
```

gens()

Return tuple of generators of this group.

These need not be minimal, as they are the generators used in defining this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3)], [(1,2)]])
sage: G.gens()
((1,2), (1,2,3))
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3))], [(Integer(1),
↪Integer(2))]])
>>> G.gens()
((1,2), (1,2,3))
```

Note that the generators need not be minimal, though duplicates are removed:

```
sage: G = PermutationGroup([[(1,2)], [(1,3)], [(2,3)], [(1,2)]])
sage: G.gens()
((2,3), (1,2), (1,3))
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2))], [(Integer(1),
↪Integer(3))], [(Integer(2),Integer(3))], [(Integer(1),Integer(2))]])
>>> G.gens()
((2,3), (1,2), (1,3))
```

We can use index notation to access the generators returned by `self.gens`:

```
sage: G = PermutationGroup([[(1,2,3,4), (5,6)], [(1,2)]])
sage: g = G.gens()
sage: g[0]
(1,2)
sage: g[1]
(1,2,3,4)(5,6)
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4)),␣
↪(Integer(5),Integer(6))], [(Integer(1),Integer(2))]])
>>> g = G.gens()
>>> g[Integer(0)]
(1,2)
>>> g[Integer(1)]
(1,2,3,4)(5,6)
```

gens_small()

For this group, returns a generating set which has few elements.

As neither irredundancy nor minimal length is proven, it is fast.

EXAMPLES:

```
sage: R = "(25,27,32,30)(26,29,31,28)( 3,38,43,19)( 5,36,45,21)( 8,33,48,24)"
↪## R = right
sage: U = "( 1, 3, 8, 6)( 2, 5, 7, 4)( 9,33,25,17)(10,34,26,18)(11,35,27,19)"
↪## U = top
sage: L = "( 9,11,16,14)(10,13,15,12)( 1,17,41,40)( 4,20,44,37)( 6,22,46,35)"
↪## L = left
sage: F = "(17,19,24,22)(18,21,23,20)( 6,25,43,16)( 7,28,42,13)( 8,30,41,11)"
↪## F = front
sage: B = "(33,35,40,38)(34,37,39,36)( 3, 9,46,32)( 2,12,47,29)( 1,14,48,27)"
↪## B = back or rear
sage: D = "(41,43,48,46)(42,45,47,44)(14,22,30,38)(15,23,31,39)(16,24,32,40)"
↪## D = down or bottom
sage: G = PermutationGroup([R,L,U,F,B,D])
sage: len(G.gens_small())
2
```

```
>>> from sage.all import *
>>> R = "(25,27,32,30)(26,29,31,28)( 3,38,43,19)( 5,36,45,21)( 8,33,48,24)" #
↪# R = right
>>> U = "( 1, 3, 8, 6)( 2, 5, 7, 4)( 9,33,25,17)(10,34,26,18)(11,35,27,19)" #
↪# U = top
>>> L = "( 9,11,16,14)(10,13,15,12)( 1,17,41,40)( 4,20,44,37)( 6,22,46,35)" #
↪# L = left
>>> F = "(17,19,24,22)(18,21,23,20)( 6,25,43,16)( 7,28,42,13)( 8,30,41,11)" #
↪# F = front
>>> B = "(33,35,40,38)(34,37,39,36)( 3, 9,46,32)( 2,12,47,29)( 1,14,48,27)" #
↪# B = back or rear
>>> D = "(41,43,48,46)(42,45,47,44)(14,22,30,38)(15,23,31,39)(16,24,32,40)" #
↪# D = down or bottom
>>> G = PermutationGroup([R,L,U,F,B,D])
>>> len(G.gens_small())
2
```

The output may be unpredictable, due to the use of randomized algorithms in GAP. Note that both the following answers are equally valid.

```
sage: G = PermutationGroup([[('a','b')], [('b', 'c')], [('a', 'c')]])
sage: G.gens_small() # random
[('b','c'), ('a','c','b')] ## (on 64-bit Linux)
[('a','b'), ('a','c','b')] ## (on Solaris)
sage: len(G.gens_small()) == 2 # random
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[('a','b')], [('b', 'c')], [('a', 'c')]])
>>> G.gens_small() # random
[('b','c'), ('a','c','b')] ## (on 64-bit Linux)
```

(continues on next page)

(continued from previous page)

```
[('a','b'), ('a','c','b')] ## (on Solaris)
>>> len(G.gens_small()) == Integer(2) # random
True
```

group_id()

Return the ID code of this group, which is a list of two integers.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: G.group_id()
[12, 4]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))], [(Integer(1),Integer(2))]])
>>> G.group_id()
[12, 4]
```

group_primitive_id()

Return the index of this group in the GAP database of primitive groups.

OUTPUT:

A positive integer, following GAP's conventions. A `ValueError` is raised if the group is not primitive.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3,4,5)], [(1,5),(2,4)]])
sage: G.group_primitive_id()
2
sage: G.degree()
5
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5))], [(Integer(1),Integer(5)),(Integer(2),Integer(4))]])
>>> G.group_primitive_id()
2
>>> G.degree()
5
```

From the information of the degree and the identification number, you can recover the isomorphism class of your group in the GAP database:

```
sage: H = PrimitiveGroup(5,2)
sage: G == H
False
sage: G.is_isomorphic(H)
True
```

```
>>> from sage.all import *
>>> H = PrimitiveGroup(Integer(5),Integer(2))
```

(continues on next page)

(continued from previous page)

```
>>> G == H
False
>>> G.is_isomorphic(H)
True
```

has_regular_subgroup(*return_group=False*)

Return whether the group contains a regular subgroup.

INPUT:

- `return_group` – boolean; if `True`, a regular subgroup is returned if there is one, and `None` if there isn't. When `return_group=False` (default), only a boolean indicating whether such a group exists is returned instead.

EXAMPLES:

The symmetric group on 4 elements has a regular subgroup:

```
sage: S4 = groups.permutation.Symmetric(4)
sage: S4.has_regular_subgroup()
True
sage: S4.has_regular_subgroup(return_group=True)  # random
Subgroup of (Symmetric group of order 4! as a permutation group)
 generated by [(1,3)(2,4), (1,4)(2,3)]
```

```
>>> from sage.all import *
>>> S4 = groups.permutation.Symmetric(Integer(4))
>>> S4.has_regular_subgroup()
True
>>> S4.has_regular_subgroup(return_group=True)  # random
Subgroup of (Symmetric group of order 4! as a permutation group)
 generated by [(1,3)(2,4), (1,4)(2,3)]
```

But the automorphism group of Petersen's graph does not:

```
sage: # needs sage.graphs
sage: G = graphs.PetersenGraph().automorphism_group()
sage: G.has_regular_subgroup()
False
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> G = graphs.PetersenGraph().automorphism_group()
>>> G.has_regular_subgroup()
False
```

holomorph()

The holomorph of a group as a permutation group.

The holomorph of a group G is the semidirect product $G \rtimes_{id} Aut(G)$, where id is the identity function on $Aut(G)$, the automorphism group of G.

See Wikipedia article Holomorph (mathematics)

OUTPUT: the holomorph of a given group as permutation group via a wrapping of GAP's semidirect product function

EXAMPLES:

Thomas and Wood's 'Group Tables' (Shiva Publishing, 1980) tells us that the holomorph of C_5 is the unique group of order 20 with a trivial center.

```
sage: C5 = CyclicPermutationGroup(5)
sage: A = C5.holomorph()
sage: A.order()
20
sage: A.is_abelian()
False
sage: A.center()
Subgroup generated by [()] of
 (Permutation Group with generators [(5,6,7,8,9), (1,2,4,3)(6,7,9,8)])
sage: A
Permutation Group with generators [(5,6,7,8,9), (1,2,4,3)(6,7,9,8)]
```

```
>>> from sage.all import *
>>> C5 = CyclicPermutationGroup(Integer(5))
>>> A = C5.holomorph()
>>> A.order()
20
>>> A.is_abelian()
False
>>> A.center()
Subgroup generated by [()] of
 (Permutation Group with generators [(5,6,7,8,9), (1,2,4,3)(6,7,9,8)])
>>> A
Permutation Group with generators [(5,6,7,8,9), (1,2,4,3)(6,7,9,8)]
```

Noting that the automorphism group of D_4 is itself D_4, it can easily be shown that the holomorph is indeed an internal semidirect product of these two groups.

```
sage: D4 = DihedralGroup(4)
sage: H = D4.holomorph()
sage: H.gens()
((2,3,5,8), (2,5)(3,8), (3,8)(4,7), (1,4,6,7)(2,3,5,8), (1,8)(2,7)(3,6)(4,5))
sage: G = H.subgroup([H.gens()[0],H.gens()[1],H.gens()[2]])
sage: N = H.subgroup([H.gens()[3],H.gens()[4]])
sage: N.is_normal(H)
True
sage: G.is_isomorphic(D4)
True
sage: N.is_isomorphic(D4)
True
sage: G.intersection(N)
Permutation Group with generators [()]
sage: L = [H(x)*H(y) for x in G for y in N]; L.sort()
sage: L1 = H.list(); L1.sort()
sage: L == L1
True
```

```
>>> from sage.all import *
>>> D4 = DihedralGroup(Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> H = D4.holomorph()
>>> H.gens()
((2,3,5,8), (2,5)(3,8), (3,8)(4,7), (1,4,6,7)(2,3,5,8), (1,8)(2,7)(3,6)(4,5))
>>> G = H.subgroup([H.gens()[Integer(0)],H.gens()[Integer(1)],H.
↪gens()[Integer(2)]])
>>> N = H.subgroup([H.gens()[Integer(3)],H.gens()[Integer(4)]])
>>> N.is_normal(H)
True
>>> G.is_isomorphic(D4)
True
>>> N.is_isomorphic(D4)
True
>>> G.intersection(N)
Permutation Group with generators [()]
>>> L = [H(x)*H(y) for x in G for y in N]; L.sort()
>>> L1 = H.list(); L1.sort()
>>> L == L1
True
```

Author:

- Kevin Halasz (2012-08-14)

homology(*n*, *p=0*)

Compute the group homology $H_n(G, F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime. Wraps HAP's `GroupHomology` function, written by Graham Ellis.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

AUTHORS:

- David Joyner and Graham Ellis

The example below computes $H_7(S_5, \mathbf{Z})$, $H_7(S_5, \mathbf{Z}/2\mathbf{Z})$, $H_7(S_5, \mathbf{Z}/3\mathbf{Z})$, and $H_7(S_5, \mathbf{Z}/5\mathbf{Z})$, respectively. To compute the 2-part of $H_7(S_5, \mathbf{Z})$, use the method `homology_part()`.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.homology(7)                              # optional - gap_package_hap
Multiplicative Abelian group isomorphic to C2 x C2 x C4 x C3 x C5
sage: G.homology(7,2)                              # optional - gap_package_
↪hap
Multiplicative Abelian group isomorphic to C2 x C2 x C2 x C2 x C2
sage: G.homology(7,3)                              # optional - gap_package_
↪hap
Multiplicative Abelian group isomorphic to C3
sage: G.homology(7,5)                              # optional - gap_package_
↪hap
Multiplicative Abelian group isomorphic to C5
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> G.homology(Integer(7))                              # optional - gap_
↪package_hap
```

(continues on next page)

(continued from previous page)

```
Multiplicative Abelian group isomorphic to C2 x C2 x C4 x C3 x C5
>>> G.homology(Integer(7),Integer(2))                              # optional␣
↪- gap_package_hap
Multiplicative Abelian group isomorphic to C2 x C2 x C2 x C2 x C2
>>> G.homology(Integer(7),Integer(3))                              # optional␣
↪- gap_package_hap
Multiplicative Abelian group isomorphic to C3
>>> G.homology(Integer(7),Integer(5))                              # optional␣
↪- gap_package_hap
Multiplicative Abelian group isomorphic to C5
```

REFERENCES:

- G. Ellis, "Computing group resolutions", J. Symbolic Computation. Vol.38, (2004)1077-1118 (Available at http://hamilton.nuigalway.ie/.
- D. Joyner, "A primer on computational group homology and cohomology", http://front.math.ucdavis.edu/0706.0549

homology_part(*n*, *p=0*)

Compute the p-part of the group homology $H_n(G, F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime. Wraps HAP's `Homology` function, written by Graham Ellis, applied to the p-Sylow subgroup of G.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.homology_part(7,2)                              # optional - gap_
↪package_hap
Multiplicative Abelian group isomorphic to C2 x C2 x C2 x C2 x C4
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> G.homology_part(Integer(7),Integer(2))                              #␣
↪optional - gap_package_hap
Multiplicative Abelian group isomorphic to C2 x C2 x C2 x C2 x C4
```

AUTHORS:

- David Joyner and Graham Ellis

id()

Return the ID code of this group, which is a list of two integers.

Same as *group_id()*.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: G.group_id()
[12, 4]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))], [(Integer(1),Integer(2))]])
```

(continues on next page)

(continued from previous page)

```
>>> G.group_id()
[12, 4]
```

identity()

Return the identity element of this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)]])
sage: e = G.identity(); e                          # indirect doctest
()
sage: g = G.gen(0)
sage: g*e
(1,2,3)(4,5)
sage: e*g
(1,2,3)(4,5)

sage: S = SymmetricGroup(['a','b','c'])
sage: S.identity()
()
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))]])
>>> e = G.identity(); e                          # indirect doctest
()
>>> g = G.gen(Integer(0))
>>> g*e
(1,2,3)(4,5)
>>> e*g
(1,2,3)(4,5)

>>> S = SymmetricGroup(['a','b','c'])
>>> S.identity()
()
```

intersection(*other*)

Return the permutation group that is the intersection of `self` and `other`.

INPUT:

- `other` – a permutation group

OUTPUT:

A permutation group that is the set-theoretic intersection of `self` with `other`. The groups are viewed as subgroups of a symmetric group big enough to contain both group's symbol sets. So there is no strict notion of the two groups being subgroups of a common parent.

EXAMPLES:

```
sage: H = DihedralGroup(4)

sage: K = CyclicPermutationGroup(4)
sage: H.intersection(K)
```

(continues on next page)

(continued from previous page)

```
Permutation Group with generators [(1,2,3,4)]

sage: L = DihedralGroup(5)
sage: H.intersection(L)
Permutation Group with generators [(1,4)(2,3)]

sage: M = PermutationGroup(["()"])
sage: H.intersection(M)
Permutation Group with generators [()]
```

```
>>> from sage.all import *
>>> H = DihedralGroup(Integer(4))

>>> K = CyclicPermutationGroup(Integer(4))
>>> H.intersection(K)
Permutation Group with generators [(1,2,3,4)]

>>> L = DihedralGroup(Integer(5))
>>> H.intersection(L)
Permutation Group with generators [(1,4)(2,3)]

>>> M = PermutationGroup(["()"])
>>> H.intersection(M)
Permutation Group with generators [()]
```

Some basic properties.

```
sage: H = DihedralGroup(4)
sage: L = DihedralGroup(5)
sage: H.intersection(L) == L.intersection(H)
True
sage: H.intersection(H) == H
True
```

```
>>> from sage.all import *
>>> H = DihedralGroup(Integer(4))
>>> L = DihedralGroup(Integer(5))
>>> H.intersection(L) == L.intersection(H)
True
>>> H.intersection(H) == H
True
```

The group `other` is verified as such.

```
sage: H = DihedralGroup(4)
sage: H.intersection('junk')
Traceback (most recent call last):
...
TypeError: junk is not a permutation group
```

```
>>> from sage.all import *
>>> H = DihedralGroup(Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> H.intersection('junk')
Traceback (most recent call last):
...
TypeError: junk is not a permutation group
```

irreducible_characters()

Return a list of the irreducible characters of `self`.

EXAMPLES:

```
sage: irr = SymmetricGroup(3).irreducible_characters()                  #␣
↪needs sage.rings.number_field
sage: [x.values() for x in irr]                                         #␣
↪needs sage.rings.number_field
[[1, -1, 1], [2, 0, -1], [1, 1, 1]]
```

```
>>> from sage.all import *
>>> irr = SymmetricGroup(Integer(3)).irreducible_characters()          ␣
↪     # needs sage.rings.number_field
>>> [x.values() for x in irr]                                         #␣
↪needs sage.rings.number_field
[[1, -1, 1], [2, 0, -1], [1, 1, 1]]
```

is_abelian()

Return `True` if this group is abelian.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_abelian()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_abelian()
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
>>> G.is_abelian()
False
>>> G = PermutationGroup(['(1,2,3)(4,5)'])
>>> G.is_abelian()
True
```

is_commutative()

Return `True` if this group is commutative.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_commutative()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
```

(continues on next page)

(continued from previous page)

```
sage: G.is_commutative()
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
>>> G.is_commutative()
False
>>> G = PermutationGroup(['(1,2,3)(4,5)'])
>>> G.is_commutative()
True
```

is_cyclic()

Return `True` if this group is cyclic.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_cyclic()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_cyclic()
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
>>> G.is_cyclic()
False
>>> G = PermutationGroup(['(1,2,3)(4,5)'])
>>> G.is_cyclic()
True
```

is_elementary_abelian()

Return `True` if this group is elementary abelian. An elementary abelian group is a finite abelian group, where every nontrivial element has order p, where p is a prime.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_elementary_abelian()
False
sage: G = PermutationGroup(['(1,2,3)','(4,5,6)'])
sage: G.is_elementary_abelian()
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
>>> G.is_elementary_abelian()
False
>>> G = PermutationGroup(['(1,2,3)','(4,5,6)'])
>>> G.is_elementary_abelian()
True
```

is_isomorphic(*right*)

Return True if the groups are isomorphic.

INPUT:

- self – this group
- right – a permutation group

OUTPUT: boolean; True if self and right are isomorphic groups; False otherwise

EXAMPLES:

```
sage: v = ['(1,2,3)(4,5)', '(1,2,3,4,5)']
sage: G = PermutationGroup(v)
sage: H = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_isomorphic(H)
False
sage: G.is_isomorphic(G)
True
sage: G.is_isomorphic(PermutationGroup(list(reversed(v))))
True
```

```
>>> from sage.all import *
>>> v = ['(1,2,3)(4,5)', '(1,2,3,4,5)']
>>> G = PermutationGroup(v)
>>> H = PermutationGroup(['(1,2,3)(4,5)'])
>>> G.is_isomorphic(H)
False
>>> G.is_isomorphic(G)
True
>>> G.is_isomorphic(PermutationGroup(list(reversed(v))))
True
```

is_monomial()

Return True if the group is monomial. A finite group is monomial if every irreducible complex character is induced from a linear character of a subgroup.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_monomial()
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)'])
>>> G.is_monomial()
True
```

is_nilpotent()

Return True if this group is nilpotent.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_nilpotent()
```

(continues on next page)

(continued from previous page)

```
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_nilpotent()
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
>>> G.is_nilpotent()
False
>>> G = PermutationGroup(['(1,2,3)(4,5)'])
>>> G.is_nilpotent()
True
```

is_normal(*other*)

Return `True` if this group is a normal subgroup of `other`.

EXAMPLES:

```
sage: AlternatingGroup(4).is_normal(SymmetricGroup(4))
True
sage: H = PermutationGroup(['(1,2,3)(4,5)'])
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: H.is_normal(G)
False
```

```
>>> from sage.all import *
>>> AlternatingGroup(Integer(4)).is_normal(SymmetricGroup(Integer(4)))
True
>>> H = PermutationGroup(['(1,2,3)(4,5)'])
>>> G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
>>> H.is_normal(G)
False
```

is_perfect()

Return `True` if this group is perfect. A group is perfect if it equals its derived subgroup.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_perfect()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_perfect()
False
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
>>> G.is_perfect()
False
>>> G = PermutationGroup(['(1,2,3)(4,5)'])
>>> G.is_perfect()
False
```

is_pgroup()

Return True if this group is a p-group.

A finite group is a p-group if its order is of the form p^n for a prime integer p and a nonnegative integer n.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3,4,5)'])
sage: G.is_pgroup()
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3,4,5)'])
>>> G.is_pgroup()
True
```

is_polycyclic()

Return True if this group is polycyclic. A group is polycyclic if it has a subnormal series with cyclic factors. (For finite groups, this is the same as if the group is solvable - see *is_solvable()*.)

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_polycyclic()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_polycyclic()
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
>>> G.is_polycyclic()
False
>>> G = PermutationGroup(['(1,2,3)(4,5)'])
>>> G.is_polycyclic()
True
```

is_primitive(*domain=None*)

Return True if self acts primitively on domain.

A group G acts primitively on a set S if

1. G acts transitively on S and
2. the action induces no non-trivial block system on S.

INPUT:

- domain – optional

See also

- *blocks_all()*

EXAMPLES:

By default, test for primitivity of self on its domain:

```
sage: G = PermutationGroup([[(1,2,3,4)],[(1,2)]])
sage: G.is_primitive()
True
sage: G = PermutationGroup([[(1,2,3,4)],[(2,4)]])
sage: G.is_primitive()
False
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))],
↪[(Integer(1),Integer(2))]])
>>> G.is_primitive()
True
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))],
↪[(Integer(2),Integer(4))]])
>>> G.is_primitive()
False
```

You can specify a domain on which to test primitivity:

```
sage: G = PermutationGroup([[(1,2,3,4)],[(2,4)]])
sage: G.is_primitive([1..4])
False
sage: G = PermutationGroup([[(3,4,5,6)],[(3,4)]]) #S_4 on [3..6]
sage: G.is_primitive(G.non_fixed_points())
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))],
↪[(Integer(2),Integer(4))]])
>>> G.is_primitive((ellipsis_range(Integer(1),Ellipsis,Integer(4))))
False
>>> G = PermutationGroup([[(Integer(3),Integer(4),Integer(5),Integer(6))],
↪[(Integer(3),Integer(4))]]) #S_4 on [3..6]
>>> G.is_primitive(G.non_fixed_points())
True
```

If G does not act on the domain, it always returns `False`:

```
sage: G = PermutationGroup([[(1,2,3,4)],[(2,4)]])
sage: G.is_primitive([1,2,3])
False
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))],
↪[(Integer(2),Integer(4))]])
>>> G.is_primitive([Integer(1),Integer(2),Integer(3)])
False
```

is_regular(*domain=None*)

Return `True` if `self` acts regularly on `domain`.

A group G acts regularly on a set S if

1. G acts transitively on S and

2. G acts semi-regularly on S.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: G.is_regular()
True
sage: G = PermutationGroup([[(1,2,3,4)],[(5,6)]])
sage: G.is_regular()
False
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> G.is_regular()
True
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))],
↪[(Integer(5),Integer(6))]])
>>> G.is_regular()
False
```

You can pass in a domain on which to test regularity:

```
sage: G = PermutationGroup([[(1,2,3,4)],[(5,6)]])
sage: G.is_regular([1..4])
True
sage: G.is_regular(G.non_fixed_points())
False
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))],
↪[(Integer(5),Integer(6))]])
>>> G.is_regular((ellipsis_range(Integer(1),Ellipsis,Integer(4))))
True
>>> G.is_regular(G.non_fixed_points())
False
```

is_semi_regular(*domain=None*)

Return `True` if `self` acts semi-regularly on `domain`.

A group G acts semi-regularly on a set S if the point stabilizers of S in G are trivial.

`domain` is optional and may take several forms. See examples.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: G.is_semi_regular()
True
sage: G = PermutationGroup([[(1,2,3,4)],[(5,6)]])
sage: G.is_semi_regular()
False
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> G.is_semi_regular()
```

(continues on next page)

(continued from previous page)

```
True
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))],
↪[(Integer(5),Integer(6))]])
>>> G.is_semi_regular()
False
```

You can pass in a domain to test semi-regularity:

```
sage: G = PermutationGroup([[(1,2,3,4)],[(5,6)]])
sage: G.is_semi_regular([1..4])
True
sage: G.is_semi_regular(G.non_fixed_points())
False
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))],
↪[(Integer(5),Integer(6))]])
>>> G.is_semi_regular((ellipsis_range(Integer(1),Ellipsis,Integer(4))))
True
>>> G.is_semi_regular(G.non_fixed_points())
False
```

is_simple()

Return `True` if the group is simple.

A group is simple if it has no proper normal subgroups.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_simple()
False
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)'])
>>> G.is_simple()
False
```

is_solvable()

Return `True` if the group is solvable.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_solvable()
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)'])
>>> G.is_solvable()
True
```

is_subgroup(*other*)

Return `True` if `self` is a subgroup of `other`.

EXAMPLES:

```
sage: G = AlternatingGroup(5)
sage: H = SymmetricGroup(5)
sage: G.is_subgroup(H)
True
```

```
>>> from sage.all import *
>>> G = AlternatingGroup(Integer(5))
>>> H = SymmetricGroup(Integer(5))
>>> G.is_subgroup(H)
True
```

is_supersolvable()

Return `True` if the group is supersolvable.

A finite group is supersolvable if it has a normal series with cyclic factors.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_supersolvable()
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)'])
>>> G.is_supersolvable()
True
```

is_transitive(*domain=None*)

Return `True` if `self` acts transitively on `domain`.

A group G acts transitively on set S if for all $x, y \in S$ there is some $g \in G$ such that $x^g = y$.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.is_transitive()
True
sage: G = PermutationGroup(['(1,2)(3,4)(5,6)'])
sage: G.is_transitive()
False
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> G.is_transitive()
True
>>> G = PermutationGroup(['(1,2)(3,4)(5,6)'])
>>> G.is_transitive()
False
```

```
sage: G = PermutationGroup([[(1,2,3,4,5)],[(1,2)],[(6,7)]])
sage: G.is_transitive([1,2,3,4,5])
True
sage: G.is_transitive([1..7])
False
sage: G.is_transitive(G.non_fixed_points())
False
sage: H = PermutationGroup([[(1,2,3)],[(4,5,6)]])
sage: H.is_transitive(H.non_fixed_points())
False
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5))],[(Integer(1),Integer(2))],[(Integer(6),Integer(7))]])
>>> G.is_transitive([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)])
True
>>> G.is_transitive((ellipsis_range(Integer(1),Ellipsis,Integer(7))))
False
>>> G.is_transitive(G.non_fixed_points())
False
>>> H = PermutationGroup([[(Integer(1),Integer(2),Integer(3))],[(Integer(4),
↪Integer(5),Integer(6))]])
>>> H.is_transitive(H.non_fixed_points())
False
```

If G does not act on the domain, it always returns `False`:

```
sage: G = PermutationGroup([[(1,2,3,4,5)],[(1,2)]]) #S_5 on [1..5]
sage: G.is_transitive([1,4,5])
False
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5))],[(Integer(1),Integer(2))]]) #S_5 on [1..5]
>>> G.is_transitive([Integer(1),Integer(4),Integer(5)])
False
```

Note that this differs from the definition in GAP, where `IsTransitive` returns whether the group is transitive on the set of points moved by the group.

```
sage: G = PermutationGroup([(2,3)])
sage: G.is_transitive()
False
sage: gap(G).IsTransitive()
true
```

```
>>> from sage.all import *
>>> G = PermutationGroup([(Integer(2),Integer(3))])
>>> G.is_transitive()
False
>>> gap(G).IsTransitive()
true
```

is_trivial()

Return `True` if this group is the trivial group.

A permutation group is trivial, if it consists only of the identity element, that is, if it has no generators or only trivial generators.

EXAMPLES:

```
sage: G = PermutationGroup([], domain=["a", "b", "c"])
sage: G.is_trivial()
True
sage: SymmetricGroup(0).is_trivial()
True
sage: SymmetricGroup(1).is_trivial()
True
sage: SymmetricGroup(2).is_trivial()
False
sage: DihedralGroup(1).is_trivial()
False
```

```
>>> from sage.all import *
>>> G = PermutationGroup([], domain=["a", "b", "c"])
>>> G.is_trivial()
True
>>> SymmetricGroup(Integer(0)).is_trivial()
True
>>> SymmetricGroup(Integer(1)).is_trivial()
True
>>> SymmetricGroup(Integer(2)).is_trivial()
False
>>> DihedralGroup(Integer(1)).is_trivial()
False
```

isomorphism_to(*right*)

Return an isomorphism from `self` to `right` if the groups are isomorphic, otherwise `None`.

INPUT:

- `self` – this group
- `right` – a permutation group

OUTPUT: `None`, or a morphism of permutation groups

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: H = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.isomorphism_to(H) is None
True
sage: G = PermutationGroup([(1,2,3), (2,3)])
sage: H = PermutationGroup([(1,2,4), (1,4)])
sage: G.isomorphism_to(H)  # not tested, see below
Permutation group morphism:
  From: Permutation Group with generators [(2,3), (1,2,3)]
  To:   Permutation Group with generators [(1,2,4), (1,4)]
  Defn: [(2,3), (1,2,3)] -> [(2,4), (1,2,4)]
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
>>> H = PermutationGroup(['(1,2,3)(4,5)'])
>>> G.isomorphism_to(H) is None
True
>>> G = PermutationGroup([(Integer(1),Integer(2),Integer(3)), (Integer(2),
↪Integer(3))])
>>> H = PermutationGroup([(Integer(1),Integer(2),Integer(4)), (Integer(1),
↪Integer(4))])
>>> G.isomorphism_to(H)  # not tested, see below
Permutation group morphism:
  From: Permutation Group with generators [(2,3), (1,2,3)]
  To:   Permutation Group with generators [(1,2,4), (1,4)]
  Defn: [(2,3), (1,2,3)] -> [(2,4), (1,2,4)]
```

isomorphism_type_info_simple_group()

If the group is simple, then this returns the name of the group.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(5)
sage: G.isomorphism_type_info_simple_group()
rec(
  name := "Z(5)",
  parameter := 5,
  series := "Z",
  shortname := "C5" )
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(5))
>>> G.isomorphism_type_info_simple_group()
rec(
  name := "Z(5)",
  parameter := 5,
  series := "Z",
  shortname := "C5" )
```

iteration(*algorithm='SGS'*)

Return an iterator over the elements of this group.

INPUT:

- `algorithm` – (default: `'SGS'`) either
  - `'SGS'` – using strong generating system
  - `'BFS'` – a breadth first search on the Cayley graph with respect to `self.gens()`
  - `'DFS'` – a depth first search on the Cayley graph with respect to `self.gens()`

Note

In general, the algorithm `'SGS'` is faster. Yet, for small groups, `'BFS'` and `'DFS'` might be faster.

Note

The order in which the iterator visits the elements differs in the algorithms.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2)], [(2,3)]])

sage: list(G.iteration())
[(), (1,2,3), (1,3,2), (2,3), (1,2), (1,3)]

sage: list(G.iteration(algorithm='BFS'))
[(), (2,3), (1,2), (1,2,3), (1,3,2), (1,3)]

sage: list(G.iteration(algorithm='DFS'))
[(), (1,2), (1,3,2), (1,3), (1,2,3), (2,3)]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2))], [(Integer(2),
↪Integer(3))]])

>>> list(G.iteration())
[(), (1,2,3), (1,3,2), (2,3), (1,2), (1,3)]

>>> list(G.iteration(algorithm='BFS'))
[(), (2,3), (1,2), (1,2,3), (1,3,2), (1,3)]

>>> list(G.iteration(algorithm='DFS'))
[(), (1,2), (1,3,2), (1,3), (1,2,3), (2,3)]
```

largest_moved_point()

Return the largest point moved by a permutation in this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: G.largest_moved_point()
4
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.largest_moved_point()
10
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],
↪[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> G.largest_moved_point()
4
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],
↪[(Integer(1),Integer(2),Integer(3),Integer(4),Integer(10))]])
>>> G.largest_moved_point()
10
```

```
sage: G = PermutationGroup([[('a','b','c'),('d','e')]])
sage: G.largest_moved_point()
'e'
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[('a','b','c'),('d','e')]])
>>> G.largest_moved_point()
'e'
```

Warning

The name of this function is not good; this function should be deprecated in term of degree:

```
sage: P = PermutationGroup([[1,2,3,4]])
sage: P.largest_moved_point()
4
sage: P.cardinality()
1
```

```
>>> from sage.all import *
>>> P = PermutationGroup([[Integer(1),Integer(2),Integer(3),Integer(4)]])
>>> P.largest_moved_point()
4
>>> P.cardinality()
1
```

list()

Return list of all elements of this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3,4)], [(1,2)]])
sage: G.list()
[(), (1,4)(2,3), (1,2)(3,4), (1,3)(2,4), (2,4,3), (1,4,2),
 (1,2,3), (1,3,4), (2,3,4), (1,4,3), (1,2,4), (1,3,2), (3,4),
 (1,4,2,3), (1,2), (1,3,2,4), (2,4), (1,4,3,2), (1,2,3,4),
 (1,3), (2,3), (1,4), (1,2,4,3), (1,3,4,2)]

sage: G = PermutationGroup([[('a','b')]], domain=('a', 'b')); G
Permutation Group with generators [('a','b')]
sage: G.list()
[(), ('a','b')]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))],
↪[(Integer(1),Integer(2))]])
>>> G.list()
[(), (1,4)(2,3), (1,2)(3,4), (1,3)(2,4), (2,4,3), (1,4,2),
 (1,2,3), (1,3,4), (2,3,4), (1,4,3), (1,2,4), (1,3,2), (3,4),
 (1,4,2,3), (1,2), (1,3,2,4), (2,4), (1,4,3,2), (1,2,3,4),
 (1,3), (2,3), (1,4), (1,2,4,3), (1,3,4,2)]
```

(continues on next page)

(continued from previous page)

```
>>> G = PermutationGroup([[('a','b')]], domain=('a', 'b')); G
Permutation Group with generators [('a','b')]
>>> G.list()
[(), ('a','b')]
```

lower_central_series()

Return the lower central series of this group as a list of permutation groups.

EXAMPLES:

These computations use pseudo-random numbers, so we set the seed for reproducible testing.

```
sage: set_random_seed(0)
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.lower_central_series()
[Subgroup generated by [(3,4), (1,2,3)(4,5)] of
  (Permutation Group with generators [(3,4), (1,2,3)(4,5)]),
 Subgroup generated by [(1,5)(3,4), (1,5)(2,4), (1,3,5)] of
  (Permutation Group with generators [(3,4), (1,2,3)(4,5)])]
```

```
>>> from sage.all import *
>>> set_random_seed(Integer(0))
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
→Integer(5))],[(Integer(3),Integer(4))]])
>>> G.lower_central_series()
[Subgroup generated by [(3,4), (1,2,3)(4,5)] of
  (Permutation Group with generators [(3,4), (1,2,3)(4,5)]),
 Subgroup generated by [(1,5)(3,4), (1,5)(2,4), (1,3,5)] of
  (Permutation Group with generators [(3,4), (1,2,3)(4,5)])]
```

maximal_normal_subgroups()

Return the maximal proper normal subgroups of `self`.

This raises an error if $G/[G, G]$ is infinite, yielding infinitely many maximal normal subgroups.

EXAMPLES:

```
sage: G = PermutationGroup([(1,2,3),(4,5)])
sage: G.maximal_normal_subgroups()
[Subgroup generated by [(1,2,3)] of (Permutation Group with generators [(4,5),
→ (1,2,3)]),
 Subgroup generated by [(4,5)] of (Permutation Group with generators [(4,5),␣
→(1,2,3)])]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([(Integer(1),Integer(2),Integer(3)),(Integer(4),
→Integer(5))])
>>> G.maximal_normal_subgroups()
[Subgroup generated by [(1,2,3)] of (Permutation Group with generators [(4,5),
→ (1,2,3)]),
 Subgroup generated by [(4,5)] of (Permutation Group with generators [(4,5),␣
→(1,2,3)])]
```

minimal_generating_set()

Return a minimal generating set.

EXAMPLES:

```
sage: # needs sage.graphs
sage: g = graphs.CompleteGraph(4)
sage: g.relabel(['a','b','c','d'])
sage: mgs = g.automorphism_group().minimal_generating_set(); len(mgs)
2
sage: mgs # random
[('b','d','c'), ('a','c','b','d')]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> g = graphs.CompleteGraph(Integer(4))
>>> g.relabel(['a','b','c','d'])
>>> mgs = g.automorphism_group().minimal_generating_set(); len(mgs)
2
>>> mgs # random
[('b','d','c'), ('a','c','b','d')]
```

minimal_normal_subgroups()

Return the nontrivial minimal normal subgroups `self`.

EXAMPLES:

```
sage: G = PermutationGroup([(1,2,3),(4,5)])
sage: G.minimal_normal_subgroups()
[Subgroup generated by [(4,5)] of (Permutation Group with generators [(4,5),␣
↪(1,2,3)]),
 Subgroup generated by [(1,2,3)] of (Permutation Group with generators [(4,5),
↪ (1,2,3)])]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))])
>>> G.minimal_normal_subgroups()
[Subgroup generated by [(4,5)] of (Permutation Group with generators [(4,5),␣
↪(1,2,3)]),
 Subgroup generated by [(1,2,3)] of (Permutation Group with generators [(4,5),
↪ (1,2,3)])]
```

molien_series()

Return the Molien series of a permutation group. The function

$$M(x) = (1/|G|) \sum_{g \in G} \det(1 - x * g)^{-1}$$

is sometimes called the "Molien series" of G. GAP's `MolienSeries` is associated to a character of a group G. How are these related? A group G, given as a permutation group on n points, has a "natural" representation of dimension n, given by permutation matrices. The Molien series of G is the one associated to that permutation representation of G using the above formula. Character values then count fixed points of the corresponding permutations.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.molien_series()
-1/(x^15 - x^14 - x^13 + x^10 + x^9 + x^8 - x^7 - x^6 - x^5 + x^2 + x - 1)
sage: G = SymmetricGroup(3)
sage: G.molien_series()
-1/(x^6 - x^5 - x^4 + x^2 + x - 1)
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> G.molien_series()
-1/(x^15 - x^14 - x^13 + x^10 + x^9 + x^8 - x^7 - x^6 - x^5 + x^2 + x - 1)
>>> G = SymmetricGroup(Integer(3))
>>> G.molien_series()
-1/(x^6 - x^5 - x^4 + x^2 + x - 1)
```

Some further tests (after Issue #15817):

```
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: S4ms = SymmetricGroup(4).molien_series()
sage: G.molien_series() / S4ms
x^5 + 2*x^4 + x^3 + x^2 + 1
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> S4ms = SymmetricGroup(Integer(4)).molien_series()
>>> G.molien_series() / S4ms
x^5 + 2*x^4 + x^3 + x^2 + 1
```

This works for not-transitive groups:

```
sage: G = PermutationGroup([[(1,2)],[(3,4)]])
sage: G.molien_series() / S4ms
x^4 + x^3 + 2*x^2 + x + 1
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2))],[(Integer(3),
↪Integer(4))]])
>>> G.molien_series() / S4ms
x^4 + x^3 + 2*x^2 + x + 1
```

This works for groups with fixed points:

```
sage: G = PermutationGroup([[(2,)]])
sage: G.molien_series()
1/(x^2 - 2*x + 1)
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(2),)]])
>>> G.molien_series()
1/(x^2 - 2*x + 1)
```

ngens()

Return the number of generators of `self`.

EXAMPLES:

```
sage: A4 = PermutationGroup([[(1,2,3)], [(2,3,4)]]); A4
Permutation Group with generators [(2,3,4), (1,2,3)]
sage: A4.ngens()
2
```

```
>>> from sage.all import *
>>> A4 = PermutationGroup([[(Integer(1),Integer(2),Integer(3))], [(Integer(2),
↪Integer(3),Integer(4))]]); A4
Permutation Group with generators [(2,3,4), (1,2,3)]
>>> A4.ngens()
2
```

non_fixed_points()

Return the list of points not fixed by `self`, i.e., the subset of `self.domain()` moved by some element of `self`.

EXAMPLES:

```
sage: G = PermutationGroup([[(3,4,5)],[(7,10)]])
sage: G.non_fixed_points()
[3, 4, 5, 7, 10]
sage: G = PermutationGroup([[(2,3,6)],[(9,)]]) # note: 9 is fixed
sage: G.non_fixed_points()
[2, 3, 6]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(3),Integer(4),Integer(5))],[(Integer(7),
↪Integer(10))]])
>>> G.non_fixed_points()
[3, 4, 5, 7, 10]
>>> G = PermutationGroup([[(Integer(2),Integer(3),Integer(6))],[(Integer(9),
↪)]]) # note: 9 is fixed
>>> G.non_fixed_points()
[2, 3, 6]
```

normal_subgroups()

Return the normal subgroups of this group as a (sorted in increasing order) list of permutation groups.

The normal subgroups of $H = PSL(2,7) \times PSL(2,7)$ are 1, two copies of $PSL(2,7)$ and H itself, as the following example shows.

EXAMPLES:

```
sage: G = PSL(2,7)
sage: D = G.direct_product(G)
sage: H = D[0]
sage: NH = H.normal_subgroups()
sage: len(NH)
4
sage: NH[1].is_isomorphic(G)
True
sage: NH[2].is_isomorphic(G)
True
```

```
>>> from sage.all import *
>>> G = PSL(Integer(2),Integer(7))
>>> D = G.direct_product(G)
>>> H = D[Integer(0)]
>>> NH = H.normal_subgroups()
>>> len(NH)
4
>>> NH[Integer(1)].is_isomorphic(G)
True
>>> NH[Integer(2)].is_isomorphic(G)
True
```

normalizer(*g*)

Return the normalizer of `g` in `self`.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: g = G([(1,3)])
sage: G.normalizer(g)
Subgroup generated by [(1,3), (2,4)] of
 (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
sage: g = G([(1,2,3,4)])
sage: G.normalizer(g)
Subgroup generated by  [(1,2,3,4), (1,3)(2,4), (2,4)] of
 (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
sage: H = G.subgroup([G([(1,2,3,4)])])
sage: G.normalizer(H)
Subgroup generated by [(1,2,3,4), (1,3)(2,4), (2,4)] of
 (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],
↪[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> g = G([(Integer(1),Integer(3))])
>>> G.normalizer(g)
Subgroup generated by [(1,3), (2,4)] of
 (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
>>> g = G([(Integer(1),Integer(2),Integer(3),Integer(4))])
>>> G.normalizer(g)
Subgroup generated by  [(1,2,3,4), (1,3)(2,4), (2,4)] of
 (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
>>> H = G.subgroup([G([(Integer(1),Integer(2),Integer(3),Integer(4))])])
>>> G.normalizer(H)
Subgroup generated by [(1,2,3,4), (1,3)(2,4), (2,4)] of
 (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
```

normalizes(*other*)

Return `True` if the group `other` is normalized by `self`.

Wraps GAP's `IsNormal` function.

A group G normalizes a group U if and only if for every $g \in G$ and $u \in U$ the element u^g is a member of U. Note that U need not be a subgroup of G.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: H = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: H.normalizes(G)
False
sage: G = SymmetricGroup(3)
sage: H = PermutationGroup( [ (4,5,6) ] )
sage: G.normalizes(H)
True
sage: H.normalizes(G)
True
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)'])
>>> H = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
>>> H.normalizes(G)
False
>>> G = SymmetricGroup(Integer(3))
>>> H = PermutationGroup( [ (Integer(4),Integer(5),Integer(6)) ] )
>>> G.normalizes(H)
True
>>> H.normalizes(G)
True
```

In the last example, G and H are disjoint, so each normalizes the other.

one()

Return the identity element of this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)]])
sage: e = G.identity(); e                           # indirect doctest
()
sage: g = G.gen(0)
sage: g*e
(1,2,3)(4,5)
sage: e*g
(1,2,3)(4,5)

sage: S = SymmetricGroup(['a','b','c'])
sage: S.identity()
()
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))]])
>>> e = G.identity(); e                           # indirect doctest
()
>>> g = G.gen(Integer(0))
>>> g*e
(1,2,3)(4,5)
>>> e*g
```

(continues on next page)

(continued from previous page)

```
(1,2,3)(4,5)

>>> S = SymmetricGroup(['a','b','c'])
>>> S.identity()
()
```

orbit(*point*, *action='OnPoints'*)

Return the orbit of a point under a group action.

INPUT:

- `point` – can be a point or any of the list above, depending on the action to be considered
- `action` – string; if `point` is an element from the domain, a tuple of elements of the domain, a tuple of tuples […], this variable describes how the group is acting

  The actions currently available through this method are `'OnPoints'`, `'OnTuples'`, `'OnSets'`, `'OnPairs'`, `'OnSetsSets'`, `'OnSetsDisjointSets'`, `'OnSetsTuples'`, `'OnTuplesSets'`, `'OnTuplesTuples'`. They are taken from GAP's list of group actions, see `gap.help('Group Actions')`.

  It is set to `'OnPoints'` by default. See below for examples.

OUTPUT:

The orbit of `point` as a tuple. Each entry is an image under the action of the permutation group, if necessary converted to the corresponding container. That is, if `action='OnSets'` then each entry will be a set even if `point` was given by a list/tuple/iterable.

EXAMPLES:

```
sage: G = PermutationGroup([ [(3,4)], [(1,3)] ])
sage: G.orbit(3)
(3, 4, 1)
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.orbit(3)
(3, 4, 10, 1, 2)
sage: G = PermutationGroup([ [('c','d')], [('a','c')] ])
sage: G.orbit('a')
('a', 'c', 'd')
```

```
>>> from sage.all import *
>>> G = PermutationGroup([ [(Integer(3),Integer(4))], [(Integer(1),
↪Integer(3))] ])
>>> G.orbit(Integer(3))
(3, 4, 1)
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],␣
↪[(Integer(1),Integer(2),Integer(3),Integer(4),Integer(10))]])
>>> G.orbit(Integer(3))
(3, 4, 10, 1, 2)
>>> G = PermutationGroup([ [('c','d')], [('a','c')] ])
>>> G.orbit('a')
('a', 'c', 'd')
```

Action of S_3 on sets:

```
sage: S3 = groups.permutation.Symmetric(3)
sage: S3.orbit((1,2), action = "OnSets")
({1, 2}, {2, 3}, {1, 3})
```

```
>>> from sage.all import *
>>> S3 = groups.permutation.Symmetric(Integer(3))
>>> S3.orbit((Integer(1),Integer(2)), action = "OnSets")
({1, 2}, {2, 3}, {1, 3})
```

On tuples:

```
sage: S3.orbit((1,2), action = "OnTuples")
((1, 2), (2, 3), (2, 1), (3, 1), (1, 3), (3, 2))
```

```
>>> from sage.all import *
>>> S3.orbit((Integer(1),Integer(2)), action = "OnTuples")
((1, 2), (2, 3), (2, 1), (3, 1), (1, 3), (3, 2))
```

Action of S_4 on sets of disjoint sets:

```
sage: S4 = groups.permutation.Symmetric(4)
sage: O = S4.orbit(((1,2),(3,4)), action='OnSetsDisjointSets')
sage: {1, 2} in O[0] and {3, 4} in O[0]
True
sage: {1, 4} in O[1] and {2, 3} in O[1]
True
sage: all(x[0].union(x[1]) == {1,2,3,4} for x in O)
True
```

```
>>> from sage.all import *
>>> S4 = groups.permutation.Symmetric(Integer(4))
>>> O = S4.orbit(((Integer(1),Integer(2)),(Integer(3),Integer(4))), action=
↪'OnSetsDisjointSets')
>>> {Integer(1), Integer(2)} in O[Integer(0)] and {Integer(3), Integer(4)} in␣
↪O[Integer(0)]
True
>>> {Integer(1), Integer(4)} in O[Integer(1)] and {Integer(2), Integer(3)} in␣
↪O[Integer(1)]
True
>>> all(x[Integer(0)].union(x[Integer(1)]) == {Integer(1),Integer(2),
↪Integer(3),Integer(4)} for x in O)
True
```

Action of S_4 (on a nonstandard domain) on tuples of sets:

```
sage: S4 = PermutationGroup([ [('c','d')], [('a','c')], [('a','b')] ])
sage: orb = S4.orbit((('a','c'),('b','d')), "OnTuplesSets")
sage: expect = (({'a', 'c'}, {'b', 'd'}), ({'a', 'd'}, {'c', 'b'}),
....:           ({'c', 'b'}, {'a', 'd'}), ({'b', 'd'}, {'a', 'c'}),
....:           ({'c', 'd'}, {'a', 'b'}), ({'a', 'b'}, {'c', 'd'}))
sage: expect == orb
True
```

```
>>> from sage.all import *
>>> S4 = PermutationGroup([ [('c','d')], [('a','c')], [('a','b')] ])
>>> orb = S4.orbit((('a','c'),('b','d')), "OnTuplesSets")
>>> expect = (({'a', 'c'}, {'b', 'd'}), ({'a', 'd'}, {'c', 'b'}),
...           ({'c', 'b'}, {'a', 'd'}), ({'b', 'd'}, {'a', 'c'}),
...           ({'c', 'd'}, {'a', 'b'}), ({'a', 'b'}, {'c', 'd'}))
>>> expect == orb
True
```

Action of S_4 (on a very nonstandard domain) on tuples of sets:

```
sage: S4 = PermutationGroup([ [((11,(12,13)),'d')],
....:         [((12,(12,11)),(11,(12,13)))], [((12,(12,11)),'b')] ])
sage: orb = S4.orbit((( (11,(12,13)), (12,(12,11))),('b','d')),
....:                "OnTuplesSets")
sage: expect = (({(11, (12, 13)), (12, (12, 11))}, {'b', 'd'}),
....:           ({'d', (12, (12, 11))}, {(11, (12, 13)), 'b'}),
....:           ({(11, (12, 13)), 'b'}, {'d', (12, (12, 11))}),
....:           ({(11, (12, 13)), 'd'}, {'b', (12, (12, 11))}),
....:           ({'b', 'd'}, {(11, (12, 13)), (12, (12, 11))}),
....:           ({'b', (12, (12, 11))}, {(11, (12, 13)), 'd'}))
sage: all(x in orb for x in expect) and len(orb) == len(expect)
True
```

```
>>> from sage.all import *
>>> S4 = PermutationGroup([ [((Integer(11),(Integer(12),Integer(13))),'d')],
...         [((Integer(12),(Integer(12),Integer(11))),(Integer(11),
↪(Integer(12),Integer(13))))], [((Integer(12),(Integer(12),Integer(11))),'b
↪')] ])
>>> orb = S4.orbit((( (Integer(11),(Integer(12),Integer(13))), (Integer(12),
↪(Integer(12),Integer(11)))),('b','d')),
...                "OnTuplesSets")
>>> expect = (({(Integer(11), (Integer(12), Integer(13))), (Integer(12),␣
↪(Integer(12), Integer(11)))}, {'b', 'd'}),
...           ({'d', (Integer(12), (Integer(12), Integer(11)))},
↪{(Integer(11), (Integer(12), Integer(13))), 'b'}),
...           ({(Integer(11), (Integer(12), Integer(13))), 'b'}, {'d',␣
↪(Integer(12), (Integer(12), Integer(11)))}),
...           ({(Integer(11), (Integer(12), Integer(13))), 'd'}, {'b',␣
↪(Integer(12), (Integer(12), Integer(11)))}),
...           ({'b', 'd'}, {(Integer(11), (Integer(12), Integer(13))),␣
↪(Integer(12), (Integer(12), Integer(11)))}),
...           ({'b', (Integer(12), (Integer(12), Integer(11)))},
↪{(Integer(11), (Integer(12), Integer(13))), 'd'}))
>>> all(x in orb for x in expect) and len(orb) == len(expect)
True
```

orbits()

Return the orbits of the elements of the domain under the default group action.

EXAMPLES:

```
sage: G = PermutationGroup([ [(3,4)], [(1,3)] ])
```

(continues on next page)

(continued from previous page)

```
sage: G.orbits()
((1, 3, 4), (2,))
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.orbits()
((1, 2, 3, 4, 10), (5,), (6,), (7,), (8,), (9,))

sage: G = PermutationGroup([ [('c','d')], [('a','c')],[('b',)]])
sage: G.orbits()
(('a', 'c', 'd'), ('b',))
```

```
>>> from sage.all import *
>>> G = PermutationGroup([ [(Integer(3),Integer(4))], [(Integer(1),
→Integer(3))] ])
>>> G.orbits()
((1, 3, 4), (2,))
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],␣
→[(Integer(1),Integer(2),Integer(3),Integer(4),Integer(10))]])
>>> G.orbits()
((1, 2, 3, 4, 10), (5,), (6,), (7,), (8,), (9,))

>>> G = PermutationGroup([ [('c','d')], [('a','c')],[('b',)]])
>>> G.orbits()
(('a', 'c', 'd'), ('b',))
```

The answer is cached:

```
sage: G.orbits() is G.orbits()
True
```

```
>>> from sage.all import *
>>> G.orbits() is G.orbits()
True
```

AUTHORS:

- Nathan Dunfield

order()

Return the number of elements of this group.

See also: *degree()*.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: G.order()
12
sage: G = PermutationGroup([()])
sage: G.order()
1
sage: G = PermutationGroup([])
sage: G.order()
1
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))], [(Integer(1),Integer(2))]])
>>> G.order()
12
>>> G = PermutationGroup([()])
>>> G.order()
1
>>> G = PermutationGroup([])
>>> G.order()
1
```

`cardinality()` is just an alias:

```
sage: PermutationGroup([(1,2,3)]).cardinality()
3
```

```
>>> from sage.all import *
>>> PermutationGroup([(Integer(1),Integer(2),Integer(3))]).cardinality()
3
```

poincare_series(*p=2*, *n=10*)

Return the Poincaré series of $G \mod p$ ($p \geq 2$ must be a prime), for n large.

In other words, if you input a finite group G, a prime p, and a positive integer n, it returns a quotient of polynomials $f(x) = P(x)/Q(x)$ whose coefficient of x^k equals the rank of the vector space $H_k(G, \mathbf{Z}/p\mathbf{Z})$, for all k in the range $1 \leq k \leq n$.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.poincare_series(2, 10)                              # optional - gap_
↪package_hap
(x^2 + 1)/(x^4 - x^3 - x + 1)
sage: G = SymmetricGroup(3)
sage: G.poincare_series(2, 10)                              # optional - gap_
↪package_hap
-1/(x - 1)
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> G.poincare_series(Integer(2), Integer(10))                              #␣
↪optional - gap_package_hap
(x^2 + 1)/(x^4 - x^3 - x + 1)
>>> G = SymmetricGroup(Integer(3))
>>> G.poincare_series(Integer(2), Integer(10))                              #␣
↪optional - gap_package_hap
-1/(x - 1)
```

AUTHORS:

- David Joyner and Graham Ellis

quotient(*N*, ***kwds*)

Return the quotient of this permutation group by the normal subgroup N, as a permutation group.

Further named arguments are passed to the permutation group constructor.

Wraps the GAP operator "/".

EXAMPLES:

```
sage: G = PermutationGroup([(1,2,3), (2,3)])
sage: N = PermutationGroup([(1,2,3)])
sage: G.quotient(N)
Permutation Group with generators [(1,2)]
sage: G.quotient(G)
Permutation Group with generators [(), ()]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([(Integer(1),Integer(2),Integer(3)), (Integer(2),
↪Integer(3))])
>>> N = PermutationGroup([(Integer(1),Integer(2),Integer(3))])
>>> G.quotient(N)
Permutation Group with generators [(1,2)]
>>> G.quotient(G)
Permutation Group with generators [(), ()]
```

random_element()

Return a random element of this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: a = G.random_element()
sage: a in G
True
sage: a.parent() is G
True
sage: a^6
()
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))], [(Integer(1),Integer(2))]])
>>> a = G.random_element()
>>> a in G
True
>>> a.parent() is G
True
>>> a**Integer(6)
()
```

representative_action(*x*, *y*)

Return an element of `self` that maps x to y if it exists.

This method wraps the gap function `RepresentativeAction`, which can also return elements that map a given set of points on another set of points.

INPUT:

- `x`, `y` – two elements of the domain

EXAMPLES:

```
sage: G = groups.permutation.Cyclic(14)
sage: g = G.representative_action(1, 10)
sage: all(g(x) == 1 + ((x+9-1)%14) for x in G.domain())
True
```

```
>>> from sage.all import *
>>> G = groups.permutation.Cyclic(Integer(14))
>>> g = G.representative_action(Integer(1), Integer(10))
>>> all(g(x) == Integer(1) + ((x+Integer(9)-Integer(1))%Integer(14)) for x in␣
↪G.domain())
True
```

semidirect_product(*N*, *mapping*, *check=True*)

The semidirect product of `self` with `N`.

INPUT:

- `N` – a group which is acted on by `self` and naturally embeds as a normal subgroup of the returned semidirect product
- `mapping` – a pair of lists that together define a homomorphism, ϕ : `self` $\rightarrow$ Aut(`N`), by giving, in the second list, the images of the generators of `self` in the order given in the first list
- `check` – a boolean that, if set to `False`, will skip the initial tests which are made on `mapping`. This may be beneficial for large `N`, since in such cases the injectivity test can be expensive. Set to `True` by default.

OUTPUT:

The semidirect product of `self` and `N` defined by the action of `self` on `N` given in `mapping` (note that a homomorphism from A to the automorphism group of B is equivalent to an action of A on B's underlying set). The semidirect product of two groups, H and N, is a construct similar to the direct product in so far as the elements are the Cartesian product of the elements of H and the elements of N. The operation, however, is built upon an action of H on N, and is defined as such:

$$(h_1, n_1)(h_2, n_2) = (h_1 h_2, n_1^{h_2} n_2)$$

This function is a wrapper for GAP's `SemidirectProduct` command. The permutation group returned is built upon a permutation representation of the semidirect product of `self` and `N` on a set of size $\mid N \mid$. The generators of `N` are given as their right regular representations, while the generators of `self` are defined by the underlying action of `self` on `N`. It should be noted that the defining action is not always faithful, and in this case the inputted representations of the generators of `self` are placed on additional letters and adjoined to the output's generators of `self`.

EXAMPLES:

Perhaps the most common example of a semidirect product comes from the family of dihedral groups. Each dihedral group is the semidirect product of C_2 with C_n, where, by convention, $3 \leq n$. In this case, the nontrivial element of C_2 acts on C_n so as to send each element to its inverse.

```
sage: C2 = CyclicPermutationGroup(2)
sage: C8 = CyclicPermutationGroup(8)
```

(continues on next page)

(continued from previous page)

```
sage: alpha = PermutationGroupMorphism_im_gens(C8,C8,[(1,8,7,6,5,4,3,2)])
sage: S = C2.semidirect_product(C8,[[(1,2)],[alpha]])
sage: S == DihedralGroup(8)
False
sage: S.is_isomorphic(DihedralGroup(8))
True
sage: S.gens()
((3,4,5,6,7,8,9,10), (1,2)(4,10)(5,9)(6,8))
```

```
>>> from sage.all import *
>>> C2 = CyclicPermutationGroup(Integer(2))
>>> C8 = CyclicPermutationGroup(Integer(8))
>>> alpha = PermutationGroupMorphism_im_gens(C8,C8,[(Integer(1),Integer(8),
↪Integer(7),Integer(6),Integer(5),Integer(4),Integer(3),Integer(2))])
>>> S = C2.semidirect_product(C8,[[(Integer(1),Integer(2))],[alpha]])
>>> S == DihedralGroup(Integer(8))
False
>>> S.is_isomorphic(DihedralGroup(Integer(8)))
True
>>> S.gens()
((3,4,5,6,7,8,9,10), (1,2)(4,10)(5,9)(6,8))
```

A more complicated example can be drawn from [TW1980]. It is there given that a semidirect product of D_4 and C_3 is isomorphic to one of C_2 and the dicyclic group of order 12. This nonabelian group of order 24 has very similar structure to the dicyclic and dihedral groups of order 24, the three being the only groups of order 24 with a two-element center and 9 conjugacy classes.

```
sage: D4 = DihedralGroup(4)
sage: C3 = CyclicPermutationGroup(3)
sage: alpha1 = PermutationGroupMorphism_im_gens(C3,C3,[(1,3,2)])
sage: alpha2 = PermutationGroupMorphism_im_gens(C3,C3,[(1,2,3)])
sage: S1 = D4.semidirect_product(C3,[[(1,2,3,4),(1,3)],[alpha1,alpha2]])
sage: C2 = CyclicPermutationGroup(2)
sage: Q = DiCyclicGroup(3)
sage: a = Q.gens()[0]; b=Q.gens()[1].inverse()
sage: alpha = PermutationGroupMorphism_im_gens(Q,Q,[a,b])
sage: S2 = C2.semidirect_product(Q,[[(1,2)],[alpha]])
sage: S1.is_isomorphic(S2)
True
sage: S1.is_isomorphic(DihedralGroup(12))
False
sage: S1.is_isomorphic(DiCyclicGroup(6))
False
sage: S1.center()
Subgroup generated by [(1,3)(2,4)] of
 (Permutation Group with generators [(5,6,7), (1,2,3,4)(6,7), (1,3)])
sage: len(S1.conjugacy_classes_representatives())
9
```

```
>>> from sage.all import *
>>> D4 = DihedralGroup(Integer(4))
>>> C3 = CyclicPermutationGroup(Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> alpha1 = PermutationGroupMorphism_im_gens(C3,C3,[(Integer(1),Integer(3),
↪Integer(2))])
>>> alpha2 = PermutationGroupMorphism_im_gens(C3,C3,[(Integer(1),Integer(2),
↪Integer(3))])
>>> S1 = D4.semidirect_product(C3,[[(Integer(1),Integer(2),Integer(3),
↪Integer(4)),(Integer(1),Integer(3))],[alpha1,alpha2]])
>>> C2 = CyclicPermutationGroup(Integer(2))
>>> Q = DiCyclicGroup(Integer(3))
>>> a = Q.gens()[Integer(0)]; b=Q.gens()[Integer(1)].inverse()
>>> alpha = PermutationGroupMorphism_im_gens(Q,Q,[a,b])
>>> S2 = C2.semidirect_product(Q,[[(Integer(1),Integer(2))],[alpha]])
>>> S1.is_isomorphic(S2)
True
>>> S1.is_isomorphic(DihedralGroup(Integer(12)))
False
>>> S1.is_isomorphic(DiCyclicGroup(Integer(6)))
False
>>> S1.center()
Subgroup generated by [(1,3)(2,4)] of
 (Permutation Group with generators [(5,6,7), (1,2,3,4)(6,7), (1,3)])
>>> len(S1.conjugacy_classes_representatives())
9
```

If your normal subgroup is large, and you are confident that your inputs will successfully create a semidirect product, then it is beneficial, for the sake of time efficiency, to set the `check` parameter to `False`.

```
sage: C2 = CyclicPermutationGroup(2)
sage: C2000 = CyclicPermutationGroup(500)
sage: alpha = PermutationGroupMorphism(C2000,C2000,[C2000.gen().inverse()])
sage: S = C2.semidirect_product(C2000,[[(1,2)],[alpha]],check=False)
```

```
>>> from sage.all import *
>>> C2 = CyclicPermutationGroup(Integer(2))
>>> C2000 = CyclicPermutationGroup(Integer(500))
>>> alpha = PermutationGroupMorphism(C2000,C2000,[C2000.gen().inverse()])
>>> S = C2.semidirect_product(C2000,[[(Integer(1),Integer(2))],[alpha]],
↪check=False)
```

AUTHOR:

- Kevin Halasz (2012-8-12)

sign_representation(*base_ring=None*)

Return the sign representation of `self` over `base_ring`.

INPUT:

- `base_ring` – (optional) the base ring; the default is $\mathbf{Z}$

EXAMPLES:

```
sage: G = groups.permutation.Dihedral(4)
sage: G.sign_representation()
Sign representation of Dihedral group of order 8
 as a permutation group over Integer Ring
```

```
>>> from sage.all import *
>>> G = groups.permutation.Dihedral(Integer(4))
>>> G.sign_representation()
Sign representation of Dihedral group of order 8
 as a permutation group over Integer Ring
```

smallest_moved_point()

Return the smallest point moved by a permutation in this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(3,4)], [(2,3,4)]])
sage: G.smallest_moved_point()
2
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.smallest_moved_point()
1
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(3),Integer(4))], [(Integer(2),Integer(3),
→Integer(4))]])
>>> G.smallest_moved_point()
2
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],␣
→[(Integer(1),Integer(2),Integer(3),Integer(4),Integer(10))]])
>>> G.smallest_moved_point()
1
```

Note that this function uses the ordering from the domain:

```
sage: S = SymmetricGroup(['a','b','c'])
sage: S.smallest_moved_point()
'a'
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(['a','b','c'])
>>> S.smallest_moved_point()
'a'
```

socle()

Return the socle of `self`.

The socle of a group G is the subgroup generated by all minimal normal subgroups.

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: s = G.socle(); s
Subgroup generated by [(1,2)(3,4), (1,4)(2,3)] of
 (Symmetric group of order 4! as a permutation group)
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> s = G.socle(); s
Subgroup generated by [(1,2)(3,4), (1,4)(2,3)] of
 (Symmetric group of order 4! as a permutation group)
```

The socle of the socle is, essentially, the socle:

```
sage: s.socle() == s.subgroup(s.gens())
True
```

```
>>> from sage.all import *
>>> s.socle() == s.subgroup(s.gens())
True
```

solvable_radical()

Return the solvable radical of `self`.

The solvable radical (or just radical) of a group G is the largest solvable normal subgroup of G.

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: G.solvable_radical()
Subgroup generated by [(1,2,3,4), (1,2)] of
 (Symmetric group of order 4! as a permutation group)
sage: G = SymmetricGroup(5)
sage: G.solvable_radical()
Subgroup generated by [()] of
 (Symmetric group of order 5! as a permutation group)
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
>>> G.solvable_radical()
Subgroup generated by [(1,2,3,4), (1,2)] of
 (Symmetric group of order 4! as a permutation group)
>>> G = SymmetricGroup(Integer(5))
>>> G.solvable_radical()
Subgroup generated by [()] of
 (Symmetric group of order 5! as a permutation group)
```

stabilizer(*point*, *action='OnPoints'*)

Return the subgroup of `self` which stabilize the given position. `self` and its stabilizers must have same degree.

INPUT:

- `point` – a point of the `domain()`, or a set of points depending on the value of `action`
- `action` – string (default: `'OnPoints'`); should the group be considered to act on points (`action="OnPoints"`) or on sets of points (`action="OnSets"`)? In the latter case, the first argument must be a subset of `domain()`.

EXAMPLES:

```
sage: G = PermutationGroup([ [(3,4)], [(1,3)] ])
sage: G.stabilizer(1)
Subgroup generated by [(3,4)] of (Permutation Group with generators [(3,4),␣
↪(1,3)])
sage: G.stabilizer(3)
Subgroup generated by [(1,4)] of (Permutation Group with generators [(3,4),␣
↪(1,3)])
```

```
>>> from sage.all import *
>>> G = PermutationGroup([ [(Integer(3),Integer(4))], [(Integer(1),
↪Integer(3))] ])
>>> G.stabilizer(Integer(1))
Subgroup generated by [(3,4)] of (Permutation Group with generators [(3,4),␣
↪(1,3)])
>>> G.stabilizer(Integer(3))
Subgroup generated by [(1,4)] of (Permutation Group with generators [(3,4),␣
↪(1,3)])
```

The stabilizer of a set of points:

```
sage: s10 = groups.permutation.Symmetric(10)
sage: s10.stabilizer([1..3],"OnSets").cardinality()
30240
sage: factorial(3)*factorial(7)
30240
```

```
>>> from sage.all import *
>>> s10 = groups.permutation.Symmetric(Integer(10))
>>> s10.stabilizer((ellipsis_range(Integer(1),Ellipsis,Integer(3))),"OnSets").
↪cardinality()
30240
>>> factorial(Integer(3))*factorial(Integer(7))
30240
```

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.stabilizer(10)
Subgroup generated by [(1,2)(3,4), (2,3,4)] of (Permutation Group with␣
↪generators [(1,2)(3,4), (1,2,3,4,10)])
sage: G.stabilizer(1) == G.subgroup(['(2,3)(4,10)', '(2,10,3)'])
True
sage: G = PermutationGroup([[(2,3,4)],[(6,7)]])
sage: G.stabilizer(1)
Subgroup generated by [(6,7), (2,3,4)] of (Permutation Group with generators␣
↪[(6,7), (2,3,4)])
sage: G.stabilizer(2)
Subgroup generated by [(6,7)] of (Permutation Group with generators [(6,7),␣
↪(2,3,4)])
sage: G.stabilizer(3)
Subgroup generated by [(6,7)] of (Permutation Group with generators [(6,7),␣
↪(2,3,4)])
sage: G.stabilizer(4)
Subgroup generated by [(6,7)] of (Permutation Group with generators [(6,7),␣
```

(continues on next page)

(continued from previous page)

```
↪(2,3,4)])
sage: G.stabilizer(5)
Subgroup generated by [(6,7), (2,3,4)] of (Permutation Group with generators␣
↪[(6,7), (2,3,4)])
sage: G.stabilizer(6)
Subgroup generated by [(2,3,4)] of (Permutation Group with generators [(6,7),␣
↪(2,3,4)])
sage: G.stabilizer(7)
Subgroup generated by [(2,3,4)] of (Permutation Group with generators [(6,7),␣
↪(2,3,4)])
sage: G.stabilizer(8)
Traceback (most recent call last):
...
ValueError: 8 does not belong to the domain
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],␣
↪[(Integer(1),Integer(2),Integer(3),Integer(4),Integer(10))]])
>>> G.stabilizer(Integer(10))
Subgroup generated by [(1,2)(3,4), (2,3,4)] of (Permutation Group with␣
↪generators [(1,2)(3,4), (1,2,3,4,10)])
>>> G.stabilizer(Integer(1)) == G.subgroup(['(2,3)(4,10)', '(2,10,3)'])
True
>>> G = PermutationGroup([[(Integer(2),Integer(3),Integer(4))],[(Integer(6),
↪Integer(7))]])
>>> G.stabilizer(Integer(1))
Subgroup generated by [(6,7), (2,3,4)] of (Permutation Group with generators␣
↪[(6,7), (2,3,4)])
>>> G.stabilizer(Integer(2))
Subgroup generated by [(6,7)] of (Permutation Group with generators [(6,7),␣
↪(2,3,4)])
>>> G.stabilizer(Integer(3))
Subgroup generated by [(6,7)] of (Permutation Group with generators [(6,7),␣
↪(2,3,4)])
>>> G.stabilizer(Integer(4))
Subgroup generated by [(6,7)] of (Permutation Group with generators [(6,7),␣
↪(2,3,4)])
>>> G.stabilizer(Integer(5))
Subgroup generated by [(6,7), (2,3,4)] of (Permutation Group with generators␣
↪[(6,7), (2,3,4)])
>>> G.stabilizer(Integer(6))
Subgroup generated by [(2,3,4)] of (Permutation Group with generators [(6,7),␣
↪(2,3,4)])
>>> G.stabilizer(Integer(7))
Subgroup generated by [(2,3,4)] of (Permutation Group with generators [(6,7),␣
↪(2,3,4)])
>>> G.stabilizer(Integer(8))
Traceback (most recent call last):
...
ValueError: 8 does not belong to the domain
```

```
sage: G = PermutationGroup([ [('c','d')], [('a','c')] ], domain='abcd')
sage: G.stabilizer('a')
Subgroup generated by [('c','d')] of (Permutation Group with generators [('c',
→'d'), ('a','c')])
sage: G.stabilizer('b')
Subgroup generated by [('c','d'), ('a','c')] of (Permutation Group with␣
→generators [('c','d'), ('a','c')])
sage: G.stabilizer('c')
Subgroup generated by [('a','d')] of (Permutation Group with generators [('c',
→'d'), ('a','c')])
sage: G.stabilizer('d')
Subgroup generated by [('a','c')] of (Permutation Group with generators [('c',
→'d'), ('a','c')])
```

```
>>> from sage.all import *
>>> G = PermutationGroup([ [('c','d')], [('a','c')] ], domain='abcd')
>>> G.stabilizer('a')
Subgroup generated by [('c','d')] of (Permutation Group with generators [('c',
→'d'), ('a','c')])
>>> G.stabilizer('b')
Subgroup generated by [('c','d'), ('a','c')] of (Permutation Group with␣
→generators [('c','d'), ('a','c')])
>>> G.stabilizer('c')
Subgroup generated by [('a','d')] of (Permutation Group with generators [('c',
→'d'), ('a','c')])
>>> G.stabilizer('d')
Subgroup generated by [('a','c')] of (Permutation Group with generators [('c',
→'d'), ('a','c')])
```

strong_generating_system(*base_of_group=None*, *implementation='sage'*)

Return a Strong Generating System of `self` according the given base for the right action of `self` on itself.

`base_of_group` is a list of the positions on which `self` acts, in any order. The algorithm returns a list of transversals and each transversal is a list of permutations. By default, `base_of_group` is `[1, 2, 3, ..., d]` where d is the degree of the group.

For `base_of_group` = $[\mathrm{pos}_1, \mathrm{pos}_2, \dots, \mathrm{pos}_d]$ let G_i be the subgroup of G = `self` which stabilizes $\mathrm{pos}_1, \mathrm{pos}_2, \dots, \mathrm{pos}_i$, so

$$G = G_0 \supset G_1 \supset G_2 \supset \cdots \supset G_n = \{e\}$$

Then the algorithm returns $[G_i.\mathrm{transversals}(\mathrm{pos}_{i+1})]_{1 \leq i \leq n}$

INPUT:

- `base_of_group` – (default: `[1, 2, 3, ..., d]`) a list containing the integers $1, 2, \dots, d$ in any order, where d is the degree of `self`
- `implementation` – (default: `'sage'`) either
  - `'sage'` – use the direct implementation in Sage
  - `'gap'` – if used, the `base_of_group` must be `None` and the computation is directly performed in GAP

OUTPUT: list of lists of permutations from the group, which forms a strong generating system

> **Warning**
>
> The outputs for implementations `'sage'` and `'gap'` differ: First, the output is reversed, and second, it might be that `'sage'` does not contain the trivial subgroup while `'gap'` does.
>
> Also, both algorithms might yield different results based on the order in which `base_of_group` is given in the first situation.

EXAMPLES:

```
sage: G = PermutationGroup([[(7,8)],[(3,4)],[(4,5)]])
sage: G.strong_generating_system()
[[()], [()], [(), (3,4), (3,5,4)], [(), (4,5)], [()], [()], [(), (7,8)], [()]]
sage: G = PermutationGroup([[(1,2,3,4)],[(1,2)]])
sage: G.strong_generating_system()
[[(), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3)],
[(), (2,4,3), (2,3,4)],
[(), (3,4)],
[()]]
sage: G = PermutationGroup([[(1,2,3)],[(4,5,7)],[(1,4,6)]])
sage: G.strong_generating_system()
[[(), (1,2,3), (1,4,6), (1,3,2), (1,5,7,4,6), (1,6,4), (1,7,5,4,6)],
 [(), (2,3,6), (2,6,3), (2,7,5,6,3), (2,5,6,3)(4,7), (2,4,5,6,3)],
 [(), (3,5,6), (3,4,7,5,6), (3,6)(5,7), (3,7,4,5,6)],
 [(), (4,7,5), (4,5,7), (4,6,7)],
 [(), (5,6,7), (5,7,6)], [()], [()]]
sage: G = PermutationGroup([[(1,2,3)],[(2,3,4)],[(3,4,5)]])
sage: G.strong_generating_system([5,4,3,2,1])
[[(), (1,5,3,4,2), (1,5,4,3,2), (1,5)(2,3), (1,5,2)],
 [(1,4)(2,3), (1,4,3), (1,4,2), ()],
 [(1,2,3), (1,3,2), ()], [()], [()]]
sage: G = PermutationGroup([[(3,4)]])
sage: G.strong_generating_system()
[[()], [()], [(), (3,4)], [()]]
sage: G.strong_generating_system(base_of_group=[3,1,2,4])
[[(), (3,4)], [()], [()], [()]]
sage: G = TransitiveGroup(12,17)
sage: G.strong_generating_system()
[[(), (1,4,11,2)(3,6,5,8)(7,10,9,12), (1,8,3,2)(4,11,10,9)(5,12,7,6),
  (1,7)(2,8)(3,9)(4,10)(5,11)(6,12), (1,12,7,2)(3,10,9,8)(4,11,6,5),
  (1,11)(2,8)(3,5)(4,10)(6,12)(7,9), (1,10,11,8)(2,3,12,5)(4,9,6,7),
  (1,3)(2,8)(4,10)(5,7)(6,12)(9,11), (1,2,3,8)(4,9,10,11)(5,6,7,12),
  (1,6,7,8)(2,3,4,9)(5,10,11,12), (1,5,9)(3,11,7), (1,9,5)(3,7,11)],
 [(), (2,6,10)(4,12,8), (2,10,6)(4,8,12)],
 [()], [()], [()], [()], [()], [()], [()], [()], [()], [()]]

sage: A = PermutationGroup([(1,2),(1,2,3,4,5,6,7,8,9)])
sage: X = A.strong_generating_system()
sage: Y = A.strong_generating_system(implementation='gap')
sage: [len(x) for x in X]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
sage: [len(y) for y in Y]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(7),Integer(8))],[(Integer(3),Integer(4))],
↪[(Integer(4),Integer(5))]])
>>> G.strong_generating_system()
[[()], [()], [(), (3,4), (3,5,4)], [(), (4,5)], [()], [()], [(), (7,8)], [()]]
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3),Integer(4))],
↪[(Integer(1),Integer(2))]])
>>> G.strong_generating_system()
[[(), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3)],
[(), (2,4,3), (2,3,4)],
[(), (3,4)],
[()]]
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3))],[(Integer(4),
↪Integer(5),Integer(7))],[(Integer(1),Integer(4),Integer(6))]])
>>> G.strong_generating_system()
[[(), (1,2,3), (1,4,6), (1,3,2), (1,5,7,4,6), (1,6,4), (1,7,5,4,6)],
 [(), (2,3,6), (2,6,3), (2,7,5,6,3), (2,5,6,3)(4,7), (2,4,5,6,3)],
 [(), (3,5,6), (3,4,7,5,6), (3,6)(5,7), (3,7,4,5,6)],
 [(), (4,7,5), (4,5,7), (4,6,7)],
 [(), (5,6,7), (5,7,6)], [()], [()]]
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3))],[(Integer(2),
↪Integer(3),Integer(4))],[(Integer(3),Integer(4),Integer(5))]])
>>> G.strong_generating_system([Integer(5),Integer(4),Integer(3),Integer(2),
↪Integer(1)])
[[(), (1,5,3,4,2), (1,5,4,3,2), (1,5)(2,3), (1,5,2)],
 [(1,4)(2,3), (1,4,3), (1,4,2), ()],
 [(1,2,3), (1,3,2), ()], [()], [()]]
>>> G = PermutationGroup([[(Integer(3),Integer(4))]])
>>> G.strong_generating_system()
[[()], [()], [(), (3,4)], [()]]
>>> G.strong_generating_system(base_of_group=[Integer(3),Integer(1),
↪Integer(2),Integer(4)])
[[(), (3,4)], [()], [()], [()]]
>>> G = TransitiveGroup(Integer(12),Integer(17))
>>> G.strong_generating_system()
[[(), (1,4,11,2)(3,6,5,8)(7,10,9,12), (1,8,3,2)(4,11,10,9)(5,12,7,6),
  (1,7)(2,8)(3,9)(4,10)(5,11)(6,12), (1,12,7,2)(3,10,9,8)(4,11,6,5),
  (1,11)(2,8)(3,5)(4,10)(6,12)(7,9), (1,10,11,8)(2,3,12,5)(4,9,6,7),
  (1,3)(2,8)(4,10)(5,7)(6,12)(9,11), (1,2,3,8)(4,9,10,11)(5,6,7,12),
  (1,6,7,8)(2,3,4,9)(5,10,11,12), (1,5,9)(3,11,7), (1,9,5)(3,7,11)],
 [(), (2,6,10)(4,12,8), (2,10,6)(4,8,12)],
 [()], [()], [()], [()], [()], [()], [()], [()], [()], [()]]

>>> A = PermutationGroup([(Integer(1),Integer(2)),(Integer(1),Integer(2),
↪Integer(3),Integer(4),Integer(5),Integer(6),Integer(7),Integer(8),
↪Integer(9))])
>>> X = A.strong_generating_system()
>>> Y = A.strong_generating_system(implementation='gap')
>>> [len(x) for x in X]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> [len(y) for y in Y]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

structure_description(*G*, *latex=False*)

Return a string that tries to describe the structure of G.

This methods wraps GAP's StructureDescription method.

For full details, including the form of the returned string and the algorithm to build it, see GAP's documentation.

INPUT:

- latex – boolean (default: False); if True, return a LaTeX formatted string

OUTPUT: string

> ⚠ **Warning**
>
> From GAP's documentation: The string returned by StructureDescription is **not** an isomorphism invariant: non-isomorphic groups can have the same string value, and two isomorphic groups in different representations can produce different strings.

EXAMPLES:

```
sage: # needs sage.groups
sage: G = CyclicPermutationGroup(6)
sage: G.structure_description()
'C6'
sage: G.structure_description(latex=True)
'C_{6}'
sage: G2 = G.direct_product(G, maps=False)
sage: LatexExpr(G2.structure_description(latex=True))
C_{6} \times C_{6}
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> G = CyclicPermutationGroup(Integer(6))
>>> G.structure_description()
'C6'
>>> G.structure_description(latex=True)
'C_{6}'
>>> G2 = G.direct_product(G, maps=False)
>>> LatexExpr(G2.structure_description(latex=True))
C_{6} \times C_{6}
```

This method is mainly intended for small groups or groups with few normal subgroups. Even then there are some surprises:

```
sage: D3 = DihedralGroup(3)                                              ␣
↪  # needs sage.groups
sage: D3.structure_description()                                         ␣
↪  # needs sage.groups
'S3'
```

```
>>> from sage.all import *
>>> D3 = DihedralGroup(Integer(3))                                       ␣
```

(continues on next page)

(continued from previous page)

```
↪          # needs sage.groups
>>> D3.structure_description()
↪# needs sage.groups
'S3'
```

We use the Sage notation for the degree of dihedral groups:

```
sage: D4 = DihedralGroup(4)
↪   # needs sage.groups
sage: D4.structure_description()
↪   # needs sage.groups
'D4'
```

```
>>> from sage.all import *
>>> D4 = DihedralGroup(Integer(4))
↪          # needs sage.groups
>>> D4.structure_description()
↪# needs sage.groups
'D4'
```

Works for finitely presented groups (Issue #17573):

```
sage: F.<x, y> = FreeGroup()
↪   # needs sage.groups
sage: G = F / [x^2*y^-1, x^3*y^2, x*y*x^-1*y^-1]
↪   # needs sage.groups
sage: G.structure_description()
↪   # needs sage.groups
'C7'
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('x', 'y',)); (x, y,) = F._first_ngens(2)# needs sage.
↪groups
>>> G = F / [x**Integer(2)*y**-Integer(1), x**Integer(3)*y**Integer(2),
↪x*y*x**-Integer(1)*y**-Integer(1)]                               # needs
↪sage.groups
>>> G.structure_description()
↪# needs sage.groups
'C7'
```

And matrix groups (Issue #17573):

```
sage: groups.matrix.GL(4,2).structure_description()
↪   # needs sage.libs.gap sage.modules
'A8'
```

```
>>> from sage.all import *
>>> groups.matrix.GL(Integer(4),Integer(2)).structure_description()
↪                  # needs sage.libs.gap sage.modules
'A8'
```

subgroup(*gens=None*, *gap_group=None*, *domain=None*, *category=None*, *canonicalize=True*, *check=True*)

Wraps the `PermutationGroup_subgroup` constructor. The argument `gens` is a list of elements of `self`.

EXAMPLES:

```
sage: G = PermutationGroup([(1,2,3),(3,4,5)])
sage: g = G((1,2,3))
sage: G.subgroup([g])
Subgroup generated by [(1,2,3)] of
 (Permutation Group with generators [(3,4,5), (1,2,3)])
```

```
>>> from sage.all import *
>>> G = PermutationGroup([(Integer(1),Integer(2),Integer(3)),(Integer(3),
↪Integer(4),Integer(5))])
>>> g = G((Integer(1),Integer(2),Integer(3)))
>>> G.subgroup([g])
Subgroup generated by [(1,2,3)] of
 (Permutation Group with generators [(3,4,5), (1,2,3)])
```

subgroups()

Return a list of all the subgroups of `self`.

OUTPUT:

Each possible subgroup of `self` is contained once in the returned list. The list is in order, according to the size of the subgroups, from the trivial subgroup with one element on through up to the whole group. Conjugacy classes of subgroups are contiguous in the list.

> **Warning**
>
> For even relatively small groups this method can take a very long time to execute, or create vast amounts of output. Likely both. Its purpose is instructional, as it can be useful for studying small groups. The 156 subgroups of the full symmetric group on 5 symbols of order 120, S_5, can be computed in about a minute on commodity hardware in 2011. The 64 subgroups of the cyclic group of order $30030 = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$ takes about twice as long.
>
> For faster results, which still exhibit the structure of the possible subgroups, use *conjugacy_classes_subgroups()*.

EXAMPLES:

```
sage: G = SymmetricGroup(3)
sage: G.subgroups()
[Subgroup generated by [()] of
  (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(2,3)] of
  (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(1,2)] of
  (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(1,3)] of
  (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(1,2,3)] of
  (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(1,2,3), (2,3)] of
  (Symmetric group of order 3! as a permutation group)]

sage: G = CyclicPermutationGroup(14)
```

(continues on next page)

(continued from previous page)

```
sage: G.subgroups()
[Subgroup generated by [()] of
  (Cyclic group of order 14 as a permutation group),
 Subgroup generated by [(1,8)(2,9)(3,10)(4,11)(5,12)(6,13)(7,14)] of
  (Cyclic group of order 14 as a permutation group),
 Subgroup generated by [(1,3,5,7,9,11,13)(2,4,6,8,10,12,14)] of
  (Cyclic group of order 14 as a permutation group),
 Subgroup generated by [(1,3,5,7,9,11,13)(2,4,6,8,10,12,14),
                        (1,2,3,4,5,6,7,8,9,10,11,12,13,14)] of
  (Cyclic group of order 14 as a permutation group)]
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> G.subgroups()
[Subgroup generated by [()] of
  (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(2,3)] of
  (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(1,2)] of
  (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(1,3)] of
  (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(1,2,3)] of
  (Symmetric group of order 3! as a permutation group),
 Subgroup generated by [(1,2,3), (2,3)] of
  (Symmetric group of order 3! as a permutation group)]

>>> G = CyclicPermutationGroup(Integer(14))
>>> G.subgroups()
[Subgroup generated by [()] of
  (Cyclic group of order 14 as a permutation group),
 Subgroup generated by [(1,8)(2,9)(3,10)(4,11)(5,12)(6,13)(7,14)] of
  (Cyclic group of order 14 as a permutation group),
 Subgroup generated by [(1,3,5,7,9,11,13)(2,4,6,8,10,12,14)] of
  (Cyclic group of order 14 as a permutation group),
 Subgroup generated by [(1,3,5,7,9,11,13)(2,4,6,8,10,12,14),
                        (1,2,3,4,5,6,7,8,9,10,11,12,13,14)] of
  (Cyclic group of order 14 as a permutation group)]
```

AUTHOR:

- Rob Beezer (2011-01-24)

sylow_subgroup(p)

Return a Sylow p-subgroup of the finite group G, where p is a prime.

This is a p-subgroup of G whose index in G is coprime to p.

Wraps the GAP function `SylowSubgroup`.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)', '(2,3)'])
sage: G.sylow_subgroup(2)
```

(continues on next page)

(continued from previous page)

```
Subgroup generated by [(2,3)] of
 (Permutation Group with generators [(2,3), (1,2,3)])
sage: G.sylow_subgroup(5)
Subgroup generated by [()] of
 (Permutation Group with generators [(2,3), (1,2,3)])
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)', '(2,3)'])
>>> G.sylow_subgroup(Integer(2))
Subgroup generated by [(2,3)] of
 (Permutation Group with generators [(2,3), (1,2,3)])
>>> G.sylow_subgroup(Integer(5))
Subgroup generated by [()] of
 (Permutation Group with generators [(2,3), (1,2,3)])
```

transversals(*point*)

If G is a permutation group acting on the set $X = \{1, 2, ..., n\}$ and H is the stabilizer subgroup of <integer>, a right (respectively left) transversal is a set containing exactly one element from each right (respectively left) coset of H. This method returns a right transversal of `self` by the stabilizer of `self` on <integer> position.

EXAMPLES:

```
sage: G = PermutationGroup([ [(3,4)], [(1,3)] ])
sage: G.transversals(1)
[(), (1,3,4), (1,4,3)]
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.transversals(1)
[(), (1,2)(3,4), (1,3,2,10,4), (1,4,2,10,3), (1,10,4,3,2)]

sage: G = PermutationGroup([ [('c','d')], [('a','c')] ])
sage: G.transversals('a')
[(), ('a','c','d'), ('a','d','c')]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([ [(Integer(3),Integer(4))], [(Integer(1),
→Integer(3))] ])
>>> G.transversals(Integer(1))
[(), (1,3,4), (1,4,3)]
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],␣
→[(Integer(1),Integer(2),Integer(3),Integer(4),Integer(10))]])
>>> G.transversals(Integer(1))
[(), (1,2)(3,4), (1,3,2,10,4), (1,4,2,10,3), (1,10,4,3,2)]

>>> G = PermutationGroup([ [('c','d')], [('a','c')] ])
>>> G.transversals('a')
[(), ('a','c','d'), ('a','d','c')]
```

trivial_character()

Return the trivial character of `self`.

EXAMPLES:

```
sage: SymmetricGroup(3).trivial_character()                                    #␣
↪needs sage.rings.number_field
Character of Symmetric group of order 3! as a permutation group
```

```
>>> from sage.all import *
>>> SymmetricGroup(Integer(3)).trivial_character()                              ␣
↪     # needs sage.rings.number_field
Character of Symmetric group of order 3! as a permutation group
```

upper_central_series()

Return the upper central series of this group as a list of permutation groups.

EXAMPLES:

These computations use pseudo-random numbers, so we set the seed for reproducible testing:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.upper_central_series()
[Subgroup generated by [()] of
 (Permutation Group with generators [(3,4), (1,2,3)(4,5)])]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))],[(Integer(3),Integer(4))]])
>>> G.upper_central_series()
[Subgroup generated by [()] of
 (Permutation Group with generators [(3,4), (1,2,3)(4,5)])]
```

class sage.groups.perm_gps.permgroup.**PermutationGroup_subgroup**(*ambient*, *gens=None*, *gap_group=None*, *domain=None*, *category=None*, *canonicalize=True*, *check=True*)

Bases: *PermutationGroup_generic*

Subgroup subclass of `PermutationGroup_generic`, so instance methods are inherited.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: gens = G.gens()
sage: H = DihedralGroup(4)
sage: H.subgroup(gens)
Subgroup generated by [(1,2,3,4)] of
 (Dihedral group of order 8 as a permutation group)
sage: K = H.subgroup(gens)
sage: K.list()
[(), (1,2,3,4), (1,3)(2,4), (1,4,3,2)]
sage: K.ambient_group()
Dihedral group of order 8 as a permutation group
sage: K.gens()
((1,2,3,4),)
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(4))
>>> gens = G.gens()
>>> H = DihedralGroup(Integer(4))
>>> H.subgroup(gens)
Subgroup generated by [(1,2,3,4)] of
 (Dihedral group of order 8 as a permutation group)
>>> K = H.subgroup(gens)
>>> K.list()
[(), (1,2,3,4), (1,3)(2,4), (1,4,3,2)]
>>> K.ambient_group()
Dihedral group of order 8 as a permutation group
>>> K.gens()
((1,2,3,4),)
```

ambient_group()

Return the ambient group related to `self`.

EXAMPLES:

An example involving the dihedral group on four elements, D_8:

```
sage: G = DihedralGroup(4)
sage: H = CyclicPermutationGroup(4)
sage: gens = H.gens()
sage: S = PermutationGroup_subgroup(G, list(gens))
sage: S.ambient_group()
Dihedral group of order 8 as a permutation group
sage: S.ambient_group() == G
True
```

```
>>> from sage.all import *
>>> G = DihedralGroup(Integer(4))
>>> H = CyclicPermutationGroup(Integer(4))
>>> gens = H.gens()
>>> S = PermutationGroup_subgroup(G, list(gens))
>>> S.ambient_group()
Dihedral group of order 8 as a permutation group
>>> S.ambient_group() == G
True
```

is_normal(*other=None*)

Return `True` if this group is a normal subgroup of `other`. If `other` is not specified, then it is assumed to be the ambient group.

EXAMPLES:

```
sage: S = SymmetricGroup(['a','b','c'])
sage: H = S.subgroup([('a', 'b', 'c')]); H
Subgroup generated by [('a','b','c')] of
 (Symmetric group of order 3! as a permutation group)
sage: H.is_normal()
True
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(['a','b','c'])
>>> H = S.subgroup([('a', 'b', 'c')]); H
Subgroup generated by [('a','b','c')] of
 (Symmetric group of order 3! as a permutation group)
>>> H.is_normal()
True
```

sage.groups.perm_gps.permgroup.**direct_product_permgroups**(*P*)

Take the direct product of the permutation groups listed in P.

EXAMPLES:

```
sage: G1 = AlternatingGroup([1,2,4,5])
sage: G2 = AlternatingGroup([3,4,6,7])
sage: D = direct_product_permgroups([G1,G2,G1])
sage: D.order()
1728
sage: D = direct_product_permgroups([G1])
sage: D == G1
True
sage: direct_product_permgroups([])
Symmetric group of order 0! as a permutation group
```

```
>>> from sage.all import *
>>> G1 = AlternatingGroup([Integer(1),Integer(2),Integer(4),Integer(5)])
>>> G2 = AlternatingGroup([Integer(3),Integer(4),Integer(6),Integer(7)])
>>> D = direct_product_permgroups([G1,G2,G1])
>>> D.order()
1728
>>> D = direct_product_permgroups([G1])
>>> D == G1
True
>>> direct_product_permgroups([])
Symmetric group of order 0! as a permutation group
```

sage.groups.perm_gps.permgroup.**from_gap_list**(*G*, *src*)

Convert a string giving a list of GAP permutations into a list of elements of G.

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup import from_gap_list
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: L = from_gap_list(G, "[(1,2,3)(4,5), (3,4)]"); L
[(1,2,3)(4,5), (3,4)]
sage: L[0].parent() is G
True
sage: L[1].parent() is G
True
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.permgroup import from_gap_list
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
```

(continues on next page)

(continued from previous page)

```
↪Integer(5))],[(Integer(3),Integer(4))]])
>>> L = from_gap_list(G, "[(1,2,3)(4,5), (3,4)]"); L
[(1,2,3)(4,5), (3,4)]
>>> L[Integer(0)].parent() is G
True
>>> L[Integer(1)].parent() is G
True
```

sage.groups.perm_gps.permgroup.**hap_decorator**(*f*)

A decorator for permutation group methods that require HAP. It checks to see that HAP is installed as well as checks that the argument p is either 0 or prime.

EXAMPLES:

```
sage: # optional - gap_package_hap
sage: from sage.groups.perm_gps.permgroup import hap_decorator
sage: def foo(self, n, p=0): print("Done")
sage: foo = hap_decorator(foo)
sage: foo(None, 3)
Done
sage: foo(None, 3, 0)
Done
sage: foo(None, 3, 5)
Done
sage: foo(None, 3, 4)
Traceback (most recent call last):
...
ValueError: p must be 0 or prime
```

```
>>> from sage.all import *
>>> # optional - gap_package_hap
>>> from sage.groups.perm_gps.permgroup import hap_decorator
>>> def foo(self, n, p=Integer(0)): print("Done")
>>> foo = hap_decorator(foo)
>>> foo(None, Integer(3))
Done
>>> foo(None, Integer(3), Integer(0))
Done
>>> foo(None, Integer(3), Integer(5))
Done
>>> foo(None, Integer(3), Integer(4))
Traceback (most recent call last):
...
ValueError: p must be 0 or prime
```

sage.groups.perm_gps.permgroup.**load_hap**()

Load the GAP hap package into the default GAP interpreter interface.

EXAMPLES:

```
sage: sage.groups.perm_gps.permgroup.load_hap()         # optional - gap_package_
↪hap
```

```
>>> from sage.all import *
>>> sage.groups.perm_gps.permgroup.load_hap()          # optional - gap_package_hap
```

27.4 “Named” Permutation groups (such as the symmetric group, S_n)

You can construct the following permutation groups:

- *SymmetricGroup*, S_n of order $n!$ (n can also be a list X of distinct positive integers, in which case it returns S_X)
- *AlternatingGroup*, A_n of order $n!/2$ (n can also be a list X of distinct positive integers, in which case it returns A_X)
- *DihedralGroup*, D_n of order $2n$
- *GeneralDihedralGroup*, $Dih(G)$, where G is an abelian group
- *CyclicPermutationGroup*, C_n of order n
- *DiCyclicGroup*, nonabelian groups of order $4m$ with a unique element of order 2
- *TransitiveGroup*, n-th` transitive group of degree d from the GAP tables of transitive groups
- `TransitiveGroups(d)`, `TransitiveGroups()`, set of all of the above
- *PrimitiveGroup*, n-th` primitive group of degree d from the GAP tables of primitive groups
- `PrimitiveGroups(d)`, `PrimitiveGroups()`, set of all of the above
- `MathieuGroup(degree)`, Mathieu group of degree 9, 10, 11, 12, 21, 22, 23, or 24.
- *KleinFourGroup*, subgroup of S_4 of order 4 which is not $C_2 \times C_2$
- *QuaternionGroup*, non-abelian group of order 8, $\{\pm 1, \pm I, \pm J, \pm K\}$
- *SplitMetacyclicGroup*, nonabelian groups of order p^m with cyclic subgroups of index p
- *SemidihedralGroup*, nonabelian 2-groups with cyclic subgroups of index 2
- `PGL(n,q)`, projective general linear group of $n \times n$ matrices over the finite field $\mathbf{F}_(q)$
- `PSL(n,q)`, projective special linear group of $n \times n$ matrices over the finite field $\mathbf{F}_(q)$
- `PSp(2n,q)`, projective symplectic linear group of $2n \times 2n$ matrices over the finite field $\mathbf{F}_(q)$
- `PSU(n,q)`, projective special unitary group of $n \times n$ matrices having coefficients in the finite field $\mathbf{F}_(q^2)$ that respect a fixed nondegenerate sesquilinear form, of determinant 1.
- `PGU(n,q)`, projective general unitary group of $n \times n$ matrices having coefficients in the finite field $\mathbf{F}_(q^2)$ that respect a fixed nondegenerate sesquilinear form, modulo the centre.
- `SuzukiGroup(q)`, Suzuki group over $\mathbf{F}_(q)$, ${}^2 B_2(2^{2k+1}) = Sz(2^{2k+1})$.
- *ComplexReflectionGroup*, the complex reflection group $G(m, p, n)$ or the exceptional complex reflection group G_m
- *SmallPermutationGroup*, a permutation realization of a group specified by its GAP id.

AUTHOR:

- David Joyner (2007-06): split from permgp.py (suggested by Nick Alexander)

REFERENCES:

- Cameron, P., Permutation Groups. New York: Cambridge University Press, 1999.

- Wielandt, H., Finite Permutation Groups. New York: Academic Press, 1964.
- Dixon, J. and Mortimer, B., Permutation Groups, Springer-Verlag, Berlin/New York, 1996.

Note

Though Suzuki groups are okay, Ree groups should *not* be wrapped as permutation groups - the construction is too slow - unless (for small values or the parameter) they are made using explicit generators.

class sage.groups.perm_gps.permgroup_named.**AlternatingGroup**(*domain=None*)

Bases: *PermutationGroup_symalt*

The alternating group of order $n!/2$, as a permutation group.

INPUT:

- n – positive integer, or list or tuple thereof

Note

This group is also available via groups.permutation.Alternating().

EXAMPLES:

```
sage: G = AlternatingGroup(6)
sage: G.order()
360
sage: G
Alternating group of order 6!/2 as a permutation group
sage: G.category()
Category of finite enumerated permutation groups
sage: TestSuite(G).run()   # long time

sage: G = AlternatingGroup([1,2,4,5])
sage: G
Alternating group of order 4!/2 as a permutation group
sage: G.domain()
{1, 2, 4, 5}
sage: G.category()
Category of finite enumerated permutation groups
sage: TestSuite(G).run()
```

```
>>> from sage.all import *
>>> G = AlternatingGroup(Integer(6))
>>> G.order()
360
>>> G
Alternating group of order 6!/2 as a permutation group
>>> G.category()
Category of finite enumerated permutation groups
>>> TestSuite(G).run()   # long time

>>> G = AlternatingGroup([Integer(1),Integer(2),Integer(4),Integer(5)])
```

(continues on next page)

(continued from previous page)

```
>>> G
Alternating group of order 4!/2 as a permutation group
>>> G.domain()
{1, 2, 4, 5}
>>> G.category()
Category of finite enumerated permutation groups
>>> TestSuite(G).run()
```

class sage.groups.perm_gps.permgroup_named.**ComplexReflectionGroup**(*m*, *p=None*, *n=None*)

Bases: *PermutationGroup_unique*

A finite complex reflection group as a permutation group.

We can realize $G(m, 1, n)$ as m copies of the symmetric group S_n with s_i for $1 \leq i < n$ acting as the usual adjacent transposition on each copy of S_n. We construct the cycle $s_n = (n, 2n, \ldots, mn)$.

We construct $G(m, p, n)$ as a subgroup of $G(m, 1, n)$ by $s_i \mapsto s_i$ for all $1 \leq i < n$,

$$s_n \mapsto s_n^{-1} s_{n-1} s_n, \qquad s_{n+1} \mapsto s_n^p.$$

Note that if $p = m$, then $s_{n+1} = 1$, in which case we do not consider it as a generator.

The exceptional complex reflection groups G_m (in the Shephard-Todd classification) are not yet implemented.

INPUT:

One of the following:

- m, p, n – positive integers to construct $G(m, p, n)$
- m – integer such that $4 \leq m \leq 37$ to construct an exceptional complex reflection G_m

Note

This group is also available via groups.permutation.ComplexReflection().

Note

The convention for the index set is for $G(m, 1, n)$ to have the complex reflection of order m correspond to s_n; i.e., $s_n^m = 1$ and $s_i^2 = 1$ for all $i < m$.

EXAMPLES:

```
sage: G = groups.permutation.ComplexReflection(3, 1, 5)
sage: G.order()
29160
sage: G
Complex reflection group G(3, 1, 5) as a permutation group
sage: G.category()
Join of Category of finite enumerated permutation groups
    and Category of finite complex reflection groups

sage: G = groups.permutation.ComplexReflection(3, 3, 4)
sage: G.cardinality()
```

(continues on next page)

(continued from previous page)

```
648
sage: s1, s2, s3, s4 = G.simple_reflections()
sage: s4*s2*s4 == s2*s4*s2
True
sage: (s4*s3*s2)^2 == (s2*s4*s3)^2
True

sage: G = groups.permutation.ComplexReflection(6, 2, 3)
sage: G.cardinality()
648
sage: s1, s2, s3, s4 = G.simple_reflections()
sage: s3^2 == G.one()
True
sage: s4^3 == G.one()
True
sage: s4 * s3 * s2 == s3 * s2 * s4
True
sage: (s3*s2*s1)^2 == (s1*s3*s2)^2
True
sage: s3 * s1 * s3 == s1 * s3 * s1
True
sage: s4 * s3 * (s2*s3)^(2-1) == s2 * s4
True

sage: G = groups.permutation.ComplexReflection(4, 2, 5)
sage: G.cardinality()
61440

sage: G = groups.permutation.ComplexReflection(4)
Traceback (most recent call last):
...
NotImplementedError: exceptional complex reflection groups are not yet implemented
```

```
>>> from sage.all import *
>>> G = groups.permutation.ComplexReflection(Integer(3), Integer(1), Integer(5))
>>> G.order()
29160
>>> G
Complex reflection group G(3, 1, 5) as a permutation group
>>> G.category()
Join of Category of finite enumerated permutation groups
    and Category of finite complex reflection groups

>>> G = groups.permutation.ComplexReflection(Integer(3), Integer(3), Integer(4))
>>> G.cardinality()
648
>>> s1, s2, s3, s4 = G.simple_reflections()
>>> s4*s2*s4 == s2*s4*s2
True
>>> (s4*s3*s2)**Integer(2) == (s2*s4*s3)**Integer(2)
True
```

(continues on next page)

(continued from previous page)

```
>>> G = groups.permutation.ComplexReflection(Integer(6), Integer(2), Integer(3))
>>> G.cardinality()
648
>>> s1, s2, s3, s4 = G.simple_reflections()
>>> s3**Integer(2) == G.one()
True
>>> s4**Integer(3) == G.one()
True
>>> s4 * s3 * s2 == s3 * s2 * s4
True
>>> (s3*s2*s1)**Integer(2) == (s1*s3*s2)**Integer(2)
True
>>> s3 * s1 * s3 == s1 * s3 * s1
True
>>> s4 * s3 * (s2*s3)**(Integer(2)-Integer(1)) == s2 * s4
True

>>> G = groups.permutation.ComplexReflection(Integer(4), Integer(2), Integer(5))
>>> G.cardinality()
61440

>>> G = groups.permutation.ComplexReflection(Integer(4))
Traceback (most recent call last):
...
NotImplementedError: exceptional complex reflection groups are not yet implemented
```

REFERENCES:

- Wikipedia article Complex_reflection_group

codegrees()

Return the codegrees of `self`.

Let G be a complex reflection group. The codegrees $d_1^* \leq d_2^* \leq \cdots \leq d_\ell^*$ of G can be defined by:

$$\prod_{i=1}^{\ell}(q - d_i^* - 1) = \sum_{g \in G} \det(g) q^{\dim(V^g)},$$

where V is the natural complex vector space that G acts on and ℓ is the `rank()`.

If $m = 1$, then we are in the special case of the symmetric group and the codegrees are $(n-2, n-3, \ldots 1, 0)$. Otherwise the codegrees are $((n-1)m, (n-2)m, \ldots, m, 0)$.

EXAMPLES:

```
sage: C = groups.permutation.ComplexReflection(4, 1, 3)
sage: C.codegrees()
(8, 4, 0)
sage: G = groups.permutation.ComplexReflection(3, 3, 4)
sage: G.codegrees()
(6, 5, 3, 0)
sage: S = groups.permutation.ComplexReflection(1, 1, 3)
sage: S.codegrees()
(1, 0)
```

```
>>> from sage.all import *
>>> C = groups.permutation.ComplexReflection(Integer(4), Integer(1),␣
↪Integer(3))
>>> C.codegrees()
(8, 4, 0)
>>> G = groups.permutation.ComplexReflection(Integer(3), Integer(3),␣
↪Integer(4))
>>> G.codegrees()
(6, 5, 3, 0)
>>> S = groups.permutation.ComplexReflection(Integer(1), Integer(1),␣
↪Integer(3))
>>> S.codegrees()
(1, 0)
```

degrees()

Return the degrees of `self`.

The degrees of a complex reflection group are the degrees of the fundamental invariants of the ring of polynomial invariants.

If $m = 1$, then we are in the special case of the symmetric group and the degrees are $(2, 3, \ldots, n, n+1)$. Otherwise the degrees are $(m, 2m, \ldots, (n-1)m, nm/p)$.

EXAMPLES:

```
sage: C = groups.permutation.ComplexReflection(4, 1, 3)
sage: C.degrees()
(4, 8, 12)
sage: G = groups.permutation.ComplexReflection(4, 2, 3)
sage: G.degrees()
(4, 6, 8)
sage: Gp = groups.permutation.ComplexReflection(4, 4, 3)
sage: Gp.degrees()
(3, 4, 8)
sage: S = groups.permutation.ComplexReflection(1, 1, 3)
sage: S.degrees()
(2, 3)
```

```
>>> from sage.all import *
>>> C = groups.permutation.ComplexReflection(Integer(4), Integer(1),␣
↪Integer(3))
>>> C.degrees()
(4, 8, 12)
>>> G = groups.permutation.ComplexReflection(Integer(4), Integer(2),␣
↪Integer(3))
>>> G.degrees()
(4, 6, 8)
>>> Gp = groups.permutation.ComplexReflection(Integer(4), Integer(4),␣
↪Integer(3))
>>> Gp.degrees()
(3, 4, 8)
>>> S = groups.permutation.ComplexReflection(Integer(1), Integer(1),␣
↪Integer(3))
>>> S.degrees()
```

(continues on next page)

(continued from previous page)

```
(2, 3)
```

Check that the product of the degrees is equal to the cardinality:

```
sage: prod(C.degrees()) == C.cardinality()
True
sage: prod(G.degrees()) == G.cardinality()
True
sage: prod(Gp.degrees()) == Gp.cardinality()
True
sage: prod(S.degrees()) == S.cardinality()
True
```

```
>>> from sage.all import *
>>> prod(C.degrees()) == C.cardinality()
True
>>> prod(G.degrees()) == G.cardinality()
True
>>> prod(Gp.degrees()) == Gp.cardinality()
True
>>> prod(S.degrees()) == S.cardinality()
True
```

index_set()

Return the index set of `self`.

EXAMPLES:

```
sage: G = groups.permutation.ComplexReflection(4, 1, 3)
sage: G.index_set()
(1, 2, 3)

sage: G = groups.permutation.ComplexReflection(1, 1, 3)
sage: G.index_set()
(1, 2)

sage: G = groups.permutation.ComplexReflection(4, 2, 3)
sage: G.index_set()
(1, 2, 3, 4)

sage: G = groups.permutation.ComplexReflection(4, 4, 3)
sage: G.index_set()
(1, 2, 3)

sage: G = groups.permutation.ComplexReflection(10, 2, 1)
sage: G.index_set()
(1,)

sage: G = groups.permutation.ComplexReflection(10, 10, 1)
sage: G.index_set()
()
```

```
>>> from sage.all import *
>>> G = groups.permutation.ComplexReflection(Integer(4), Integer(1),␣
↪Integer(3))
>>> G.index_set()
(1, 2, 3)

>>> G = groups.permutation.ComplexReflection(Integer(1), Integer(1),␣
↪Integer(3))
>>> G.index_set()
(1, 2)

>>> G = groups.permutation.ComplexReflection(Integer(4), Integer(2),␣
↪Integer(3))
>>> G.index_set()
(1, 2, 3, 4)

>>> G = groups.permutation.ComplexReflection(Integer(4), Integer(4),␣
↪Integer(3))
>>> G.index_set()
(1, 2, 3)

>>> G = groups.permutation.ComplexReflection(Integer(10), Integer(2),␣
↪Integer(1))
>>> G.index_set()
(1,)

>>> G = groups.permutation.ComplexReflection(Integer(10), Integer(10),␣
↪Integer(1))
>>> G.index_set()
()
```

simple_reflection(*i*)

Return the `i`-th simple reflection of `self`.

EXAMPLES:

```
sage: G = groups.permutation.ComplexReflection(3, 1, 4)
sage: G.simple_reflection(2)
(2,3)(6,7)(10,11)
sage: G.simple_reflection(4)
(4,8,12)

sage: G = groups.permutation.ComplexReflection(1, 1, 4)
sage: G.simple_reflections()
Finite family {1: (1,2), 2: (2,3), 3: (3,4)}

sage: G = groups.permutation.ComplexReflection(10, 2, 1)
sage: G.simple_reflections()
Finite family {1: (1,3,5,7,9)(2,4,6,8,10)}

sage: G = groups.permutation.ComplexReflection(10, 10, 1)
sage: G.simple_reflections()
Finite family {}
```

```
>>> from sage.all import *
>>> G = groups.permutation.ComplexReflection(Integer(3), Integer(1),␣
↪Integer(4))
>>> G.simple_reflection(Integer(2))
(2,3)(6,7)(10,11)
>>> G.simple_reflection(Integer(4))
(4,8,12)

>>> G = groups.permutation.ComplexReflection(Integer(1), Integer(1),␣
↪Integer(4))
>>> G.simple_reflections()
Finite family {1: (1,2), 2: (2,3), 3: (3,4)}

>>> G = groups.permutation.ComplexReflection(Integer(10), Integer(2),␣
↪Integer(1))
>>> G.simple_reflections()
Finite family {1: (1,3,5,7,9)(2,4,6,8,10)}

>>> G = groups.permutation.ComplexReflection(Integer(10), Integer(10),␣
↪Integer(1))
>>> G.simple_reflections()
Finite family {}
```

class sage.groups.perm_gps.permgroup_named.**CyclicPermutationGroup**(n)

Bases: *PermutationGroup_unique*

A cyclic group of order n, as a permutation group.

INPUT:

- n – positive integer

Note

This group is also available via groups.permutation.Cyclic().

EXAMPLES:

```
sage: G = CyclicPermutationGroup(8)
sage: G.order()
8
sage: G
Cyclic group of order 8 as a permutation group
sage: G.category()
Category of finite enumerated permutation groups
sage: TestSuite(G).run()
sage: C = CyclicPermutationGroup(10)
sage: C.is_abelian()
True
sage: C = CyclicPermutationGroup(10)
sage: C.as_AbelianGroup()
Multiplicative Abelian group isomorphic to C2 x C5
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(8))
>>> G.order()
8
>>> G
Cyclic group of order 8 as a permutation group
>>> G.category()
Category of finite enumerated permutation groups
>>> TestSuite(G).run()
>>> C = CyclicPermutationGroup(Integer(10))
>>> C.is_abelian()
True
>>> C = CyclicPermutationGroup(Integer(10))
>>> C.as_AbelianGroup()
Multiplicative Abelian group isomorphic to C2 x C5
```

as_AbelianGroup()

Return the corresponding *AbelianGroup* instance.

EXAMPLES:

```
sage: C = CyclicPermutationGroup(8)
sage: C.as_AbelianGroup()
Multiplicative Abelian group isomorphic to C8
```

```
>>> from sage.all import *
>>> C = CyclicPermutationGroup(Integer(8))
>>> C.as_AbelianGroup()
Multiplicative Abelian group isomorphic to C8
```

is_abelian()

Return `True` if this group is abelian.

EXAMPLES:

```
sage: C = CyclicPermutationGroup(8)
sage: C.is_abelian()
True
```

```
>>> from sage.all import *
>>> C = CyclicPermutationGroup(Integer(8))
>>> C.is_abelian()
True
```

is_commutative()

Return `True` if this group is commutative.

EXAMPLES:

```
sage: C = CyclicPermutationGroup(8)
sage: C.is_commutative()
True
```

```
>>> from sage.all import *
>>> C = CyclicPermutationGroup(Integer(8))
>>> C.is_commutative()
True
```

class sage.groups.perm_gps.permgroup_named.**DiCyclicGroup**(*n*)

Bases: *PermutationGroup_unique*

The dicyclic group of order $4n$, for $n \geq 2$.

INPUT:

- n – positive integer, two or greater

OUTPUT:

This is a nonabelian group similar in some respects to the dihedral group of the same order, but with far fewer elements of order 2 (it has just one). The permutation representation constructed here is based on the presentation

$$\langle a, x \mid a^{2n} = 1, x^2 = a^n, x^{-1}ax = a^{-1} \rangle$$

For $n = 2$ this is the group of quaternions $(\pm 1, \pm I, \pm J, \pm K)$, which is the nonabelian group of order 8 that is not the dihedral group D_4, the symmetries of a square. For $n = 3$ this is the nonabelian group of order 12 that is not the dihedral group D_6 nor the alternating group A_4. This group of order 12 is also the semi-direct product of C_2 by C_4, $C_3 \rtimes C_4$. [Con]

When the order of the group is a power of 2 it is known as a "generalized quaternion group."

IMPLEMENTATION:

The presentation above means every element can be written as $a^i x^j$ with $0 \leq i < 2n$, $j = 0, 1$. We code a^i as the symbol $i + 1$ and code $a^i x$ as the symbol $2n + i + 1$. The two generators are then represented using a left regular representation.

Note

This group is also available via groups.permutation.DiCyclic().

EXAMPLES:

A dicyclic group of order 384, with a large power of 2 as a divisor:

```
sage: n = 3*2^5
sage: G = DiCyclicGroup(n)
sage: G.order()
384
sage: a = G.gen(0)
sage: x = G.gen(1)
sage: a^(2*n)
()
sage: a^n==x^2
True
sage: x^-1*a*x==a^-1
True
```

```
>>> from sage.all import *
>>> n = Integer(3)*Integer(2)**Integer(5)
>>> G = DiCyclicGroup(n)
>>> G.order()
384
>>> a = G.gen(Integer(0))
>>> x = G.gen(Integer(1))
>>> a**(Integer(2)*n)
()
>>> a**n==x**Integer(2)
True
>>> x**-Integer(1)*a*x==a**-Integer(1)
True
```

A large generalized quaternion group (order is a power of 2):

```
sage: n = 2^10
sage: G = DiCyclicGroup(n)
sage: G.order()
4096
sage: a = G.gen(0)
sage: x = G.gen(1)
sage: a^(2*n)
()
sage: a^n==x^2
True
sage: x^-1*a*x==a^-1
True
```

```
>>> from sage.all import *
>>> n = Integer(2)**Integer(10)
>>> G = DiCyclicGroup(n)
>>> G.order()
4096
>>> a = G.gen(Integer(0))
>>> x = G.gen(Integer(1))
>>> a**(Integer(2)*n)
()
>>> a**n==x**Integer(2)
True
>>> x**-Integer(1)*a*x==a**-Integer(1)
True
```

Just like the dihedral group, the dicyclic group has an element whose order is half the order of the group. Unlike the dihedral group, the dicyclic group has only one element of order 2. Like the dihedral groups of even order, the center of the dicyclic group is a subgroup of order 2 (thus has the unique element of order 2 as its non-identity element).

```
sage: G = DiCyclicGroup(3*5*4)
sage: G.order()
240
sage: two = [g for g in G if g.order()==2]; two
[(1,5)(2,6)(3,7)(4,8)(9,13)(10,14)(11,15)(12,16)]
```

(continues on next page)

(continued from previous page)

```
sage: G.center().order()
2
```

```
>>> from sage.all import *
>>> G = DiCyclicGroup(Integer(3)*Integer(5)*Integer(4))
>>> G.order()
240
>>> two = [g for g in G if g.order()==Integer(2)]; two
[(1,5)(2,6)(3,7)(4,8)(9,13)(10,14)(11,15)(12,16)]
>>> G.center().order()
2
```

For small orders, we check this is really a group we do not have in Sage otherwise.

```
sage: G = DiCyclicGroup(2)
sage: H = DihedralGroup(4)
sage: G.is_isomorphic(H)
False
sage: G = DiCyclicGroup(3)
sage: H = DihedralGroup(6)
sage: K = AlternatingGroup(6)
sage: G.is_isomorphic(H) or G.is_isomorphic(K)
False
```

```
>>> from sage.all import *
>>> G = DiCyclicGroup(Integer(2))
>>> H = DihedralGroup(Integer(4))
>>> G.is_isomorphic(H)
False
>>> G = DiCyclicGroup(Integer(3))
>>> H = DihedralGroup(Integer(6))
>>> K = AlternatingGroup(Integer(6))
>>> G.is_isomorphic(H) or G.is_isomorphic(K)
False
```

AUTHOR:

- Rob Beezer (2009-10-18)

is_abelian()

Return `True` if this group is abelian.

EXAMPLES:

```
sage: D = DiCyclicGroup(12)
sage: D.is_abelian()
False
```

```
>>> from sage.all import *
>>> D = DiCyclicGroup(Integer(12))
>>> D.is_abelian()
False
```

is_commutative()

Return `True` if this group is commutative.

EXAMPLES:

```
sage: D = DiCyclicGroup(12)
sage: D.is_commutative()
False
```

```
>>> from sage.all import *
>>> D = DiCyclicGroup(Integer(12))
>>> D.is_commutative()
False
```

class sage.groups.perm_gps.permgroup_named.**DihedralGroup**(*n*)

Bases: *PermutationGroup_unique*

The Dihedral group of order $2n$ for any integer $n \geq 1$.

INPUT:

- `n` – positive integer

OUTPUT: the dihedral group of order $2n$, as a permutation group

> **Note**
>
> This group is also available via `groups.permutation.Dihedral()`.

EXAMPLES:

```
sage: DihedralGroup(1)
Dihedral group of order 2 as a permutation group

sage: DihedralGroup(2)
Dihedral group of order 4 as a permutation group
sage: DihedralGroup(2).gens()
((3,4), (1,2))

sage: DihedralGroup(5).gens()
((1,2,3,4,5), (1,5)(2,4))
sage: sorted(DihedralGroup(5))
[(), (2,5)(3,4), (1,2)(3,5), (1,2,3,4,5), (1,3)(4,5), (1,3,5,2,4),
 (1,4)(2,3), (1,4,2,5,3), (1,5,4,3,2), (1,5)(2,4)]

sage: G = DihedralGroup(6)
sage: G.order()
12
sage: G = DihedralGroup(5)
sage: G.order()
10
sage: G
Dihedral group of order 10 as a permutation group
sage: G.gens()
```

(continues on next page)

(continued from previous page)

```
((1,2,3,4,5), (1,5)(2,4))

sage: DihedralGroup(0)
Traceback (most recent call last):
...
ValueError: n must be positive
```

```
>>> from sage.all import *
>>> DihedralGroup(Integer(1))
Dihedral group of order 2 as a permutation group

>>> DihedralGroup(Integer(2))
Dihedral group of order 4 as a permutation group
>>> DihedralGroup(Integer(2)).gens()
((3,4), (1,2))

>>> DihedralGroup(Integer(5)).gens()
((1,2,3,4,5), (1,5)(2,4))
>>> sorted(DihedralGroup(Integer(5)))
[(), (2,5)(3,4), (1,2)(3,5), (1,2,3,4,5), (1,3)(4,5), (1,3,5,2,4),
 (1,4)(2,3), (1,4,2,5,3), (1,5,4,3,2), (1,5)(2,4)]

>>> G = DihedralGroup(Integer(6))
>>> G.order()
12
>>> G = DihedralGroup(Integer(5))
>>> G.order()
10
>>> G
Dihedral group of order 10 as a permutation group
>>> G.gens()
((1,2,3,4,5), (1,5)(2,4))

>>> DihedralGroup(Integer(0))
Traceback (most recent call last):
...
ValueError: n must be positive
```

class sage.groups.perm_gps.permgroup_named.**GeneralDihedralGroup**(*factors*)

Bases: *PermutationGroup_generic*

The Generalized Dihedral Group generated by the abelian group with direct factors in the input list.

INPUT:

- `factors` – list of the sizes of the cyclic factors of the abelian group being dihedralized (this will be sorted once entered)

OUTPUT:

For a given abelian group (noting that each finite abelian group can be represented as the direct product of cyclic groups), the General Dihedral Group it generates is simply the semi-direct product of the given group with C_2, where the nonidentity element of C_2 acts on the abelian group by turning each element into its inverse. In this implementation, each input abelian group will be standardized so as to act on a minimal amount of letters. This will be done by breaking the direct factors into products of p-groups, before this new set of factors is ordered from smallest to largest for complete standardization. Note that the generalized dihedral group corresponding to a cyclic group, C_n, is simply the dihedral group D_n.

EXAMPLES:

As is noted in [TW1980], $Dih(C_3 \times C_3)$ has the presentation

$$\langle a, b, c \mid a^3, b^3, c^2, ab = ba, ac = ca^{-1}, bc = cb^{-1} \rangle$$

Note also the fact, verified by [TW1980], that the dihedralization of $C_3 \times C_3$ is the only nonabelian group of order 18 with no element of order 6.

```
sage: G = GeneralDihedralGroup([3,3])
sage: G
Generalized dihedral group generated by C3 x C3
sage: G.order()
18
sage: G.gens()
((4,5,6), (2,3)(5,6), (1,2,3))
sage: a = G.gens()[2]; b = G.gens()[0]; c = G.gens()[1]
sage: a.order() == 3, b.order() == 3, c.order() == 2
(True, True, True)
sage: a*b == b*a, a*c == c*a.inverse(), b*c == c*b.inverse()
(True, True, True)
sage: G.subgroup([a,b,c]) == G
True
sage: G.is_abelian()
False
sage: all(x.order() != 6 for x in G)
True
```

```
>>> from sage.all import *
>>> G = GeneralDihedralGroup([Integer(3),Integer(3)])
>>> G
Generalized dihedral group generated by C3 x C3
>>> G.order()
18
>>> G.gens()
((4,5,6), (2,3)(5,6), (1,2,3))
>>> a = G.gens()[Integer(2)]; b = G.gens()[Integer(0)]; c = G.gens()[Integer(1)]
>>> a.order() == Integer(3), b.order() == Integer(3), c.order() == Integer(2)
(True, True, True)
>>> a*b == b*a, a*c == c*a.inverse(), b*c == c*b.inverse()
(True, True, True)
>>> G.subgroup([a,b,c]) == G
True
>>> G.is_abelian()
False
>>> all(x.order() != Integer(6) for x in G)
True
```

If all of the direct factors are C_2, then the action turning each element into its inverse is trivial, and the semi-direct product becomes a direct product.

```
sage: G = GeneralDihedralGroup([2,2,2])
sage: G.order()
16
sage: G.gens()
((7,8), (5,6), (3,4), (1,2))
sage: G.is_abelian()
True
sage: H = KleinFourGroup()
sage: G.is_isomorphic(H.direct_product(H)[0])
True
```

```
>>> from sage.all import *
>>> G = GeneralDihedralGroup([Integer(2),Integer(2),Integer(2)])
>>> G.order()
16
>>> G.gens()
((7,8), (5,6), (3,4), (1,2))
>>> G.is_abelian()
True
>>> H = KleinFourGroup()
>>> G.is_isomorphic(H.direct_product(H)[Integer(0)])
True
```

If two nonidentical input lists generate isomorphic abelian groups, then they will generate identical groups (with each direct factor broken up into its prime factors), but they will still have distinct descriptions. Note that if $\gcd(n, m) = 1$, then $C_n \times C_m \cong C_{nm}$, while the general dihedral groups generated by isomorphic abelian groups should be themselves isomorphic.

```
sage: G = GeneralDihedralGroup([6,34,46,14])
sage: H = GeneralDihedralGroup([7,17,3,46,2,2,2])
sage: G == H, G.gens() == H.gens()
(True, True)
sage: [x.order() for x in G.gens()]
[23, 17, 7, 2, 3, 2, 2, 2, 2]
sage: G
Generalized dihedral group generated by C6 x C34 x C46 x C14
sage: H
Generalized dihedral group generated by C7 x C17 x C3 x C46 x C2 x C2 x C2
```

```
>>> from sage.all import *
>>> G = GeneralDihedralGroup([Integer(6),Integer(34),Integer(46),Integer(14)])
>>> H = GeneralDihedralGroup([Integer(7),Integer(17),Integer(3),Integer(46),
↪Integer(2),Integer(2),Integer(2)])
>>> G == H, G.gens() == H.gens()
(True, True)
>>> [x.order() for x in G.gens()]
[23, 17, 7, 2, 3, 2, 2, 2, 2]
>>> G
Generalized dihedral group generated by C6 x C34 x C46 x C14
>>> H
Generalized dihedral group generated by C7 x C17 x C3 x C46 x C2 x C2 x C2
```

A cyclic input yields a Classical Dihedral Group.

```
sage: G = GeneralDihedralGroup([6])
sage: D = DihedralGroup(6)
sage: G.is_isomorphic(D)
True
```

```
>>> from sage.all import *
>>> G = GeneralDihedralGroup([Integer(6)])
>>> D = DihedralGroup(Integer(6))
>>> G.is_isomorphic(D)
True
```

A Generalized Dihedral Group will always have size twice the underlying group, be solvable (as it has an abelian subgroup with index 2), and, unless the underlying group is of the form ${C_2}^n$, be nonabelian (by the structure theorem of finite abelian groups and the fact that a semi-direct product is a direct product only when the underlying action is trivial).

```
sage: G = GeneralDihedralGroup([6,18,33,60])
sage: (6*18*33*60)*2
427680
sage: G.order()
427680
sage: G.is_solvable()
True
sage: G.is_abelian()
False
```

```
>>> from sage.all import *
>>> G = GeneralDihedralGroup([Integer(6),Integer(18),Integer(33),Integer(60)])
>>> (Integer(6)*Integer(18)*Integer(33)*Integer(60))*Integer(2)
427680
>>> G.order()
427680
>>> G.is_solvable()
True
>>> G.is_abelian()
False
```

AUTHOR:

- Kevin Halasz (2012-7-12)

class sage.groups.perm_gps.permgroup_named.**JankoGroup**(*n*)

Bases: *PermutationGroup_unique*

Janko Groups $J1$, $J2$, and $J3$. (Note that $J4$ is too big to be treated here.)

INPUT:

- n – integer among $\{1, 2, 3\}$

EXAMPLES:

```
sage: G = groups.permutation.Janko(1); G                    # optional -
↪gap_package_atlasrep internet
Janko group J1 of order 175560 as a permutation group
```

```
>>> from sage.all import *
>>> G = groups.permutation.Janko(Integer(1)); G                    #␣
↪optional - gap_package_atlasrep internet
Janko group J1 of order 175560 as a permutation group
```

class sage.groups.perm_gps.permgroup_named.**KleinFourGroup**

Bases: *PermutationGroup_unique*

The Klein 4 Group, which has order 4 and exponent 2, viewed as a subgroup of S_4.

OUTPUT: the Klein 4 group of order 4, as a permutation group of degree 4

Note

This group is also available via `groups.permutation.KleinFour()`.

EXAMPLES:

```
sage: G = KleinFourGroup(); G
The Klein 4 group of order 4, as a permutation group
sage: sorted(G)
[(), (3,4), (1,2), (1,2)(3,4)]
```

```
>>> from sage.all import *
>>> G = KleinFourGroup(); G
The Klein 4 group of order 4, as a permutation group
>>> sorted(G)
[(), (3,4), (1,2), (1,2)(3,4)]
```

AUTHOR:
– Bobby Moretti (2006-10)

class sage.groups.perm_gps.permgroup_named.**MathieuGroup**(*n*)

Bases: *PermutationGroup_unique*

The Mathieu group of degree n.

INPUT:

- `n` – positive integer in {9, 10, 11, 12, 21, 22, 23, 24}

OUTPUT: the Mathieu group of degree n, as a permutation group

Note

This group is also available via `groups.permutation.Mathieu()`.

EXAMPLES:

```
sage: G = MathieuGroup(12); G
Mathieu group of degree 12 and order 95040 as a permutation group
```

```
>>> from sage.all import *
>>> G = MathieuGroup(Integer(12)); G
Mathieu group of degree 12 and order 95040 as a permutation group
```

class sage.groups.perm_gps.permgroup_named.**PGL**(*n*, *q*, *name='a'*)

Bases: *PermutationGroup_plg*

The projective general linear groups over $\mathbf{F}_(q)$.

INPUT:

- n – positive integer; the degree
- q – prime power; the size of the ground field
- name – (default: 'a') variable name of indeterminate of finite field $\mathbf{F}_(q)$

OUTPUT: PGL(n, q)

Note

This group is also available via groups.permutation.PGL().

EXAMPLES:

```
sage: G = PGL(2,3); G
Permutation Group with generators [(3,4), (1,2,4)]
sage: print(G)
The projective general linear group of degree 2 over Finite Field of size 3
sage: G.base_ring()
Finite Field of size 3
sage: G.order()
24

sage: G = PGL(2, 9, 'b'); G                                                   #␣
↪needs sage.rings.finite_rings
Permutation Group with generators [(3,10,9,8,4,7,6,5), (1,2,4)(5,6,8)(7,9,10)]
sage: G.base_ring()
Finite Field in b of size 3^2

sage: G.category()
Category of finite enumerated permutation groups
sage: TestSuite(G).run()   # long time
```

```
>>> from sage.all import *
>>> G = PGL(Integer(2),Integer(3)); G
Permutation Group with generators [(3,4), (1,2,4)]
>>> print(G)
The projective general linear group of degree 2 over Finite Field of size 3
>>> G.base_ring()
Finite Field of size 3
>>> G.order()
24
```

(continues on next page)

(continued from previous page)

```
>>> G = PGL(Integer(2), Integer(9), 'b'); G                                     ␣
↪           # needs sage.rings.finite_rings
Permutation Group with generators [(3,10,9,8,4,7,6,5), (1,2,4)(5,6,8)(7,9,10)]
>>> G.base_ring()
Finite Field in b of size 3^2

>>> G.category()
Category of finite enumerated permutation groups
>>> TestSuite(G).run()  # long time
```

class sage.groups.perm_gps.permgroup_named.**PGU**(*n*, *q*, *name='a'*)

Bases: *PermutationGroup_pug*

The projective general unitary groups over $\mathbf{F}_(q)$.

INPUT:

- n – positive integer; the degree
- q – prime power; the size of the ground field
- name – (default: 'a') variable name of indeterminate of finite field $\mathbf{F}_(q)$

OUTPUT: PGU(n, q)

Note

This group is also available via groups.permutation.PGU().

EXAMPLES:

```
sage: PGU(2,3)                                                              #␣
↪needs sage.rings.finite_rings
The projective general unitary group of degree 2 over Finite Field of size 3

sage: G = PGU(2, 8, name='alpha'); G                                        #␣
↪needs sage.rings.finite_rings
The projective general unitary group of degree 2
 over Finite Field in alpha of size 2^3
sage: G.base_ring()                                                         #␣
↪needs sage.rings.finite_rings
Finite Field in alpha of size 2^3
```

```
>>> from sage.all import *
>>> PGU(Integer(2),Integer(3))                                                  ␣
↪           # needs sage.rings.finite_rings
The projective general unitary group of degree 2 over Finite Field of size 3

>>> G = PGU(Integer(2), Integer(8), name='alpha'); G                            ␣
↪           # needs sage.rings.finite_rings
The projective general unitary group of degree 2
 over Finite Field in alpha of size 2^3
>>> G.base_ring()                                                        # needs␣
```

(continues on next page)

(continued from previous page)

```
↪sage.rings.finite_rings
Finite Field in alpha of size 2^3
```

class sage.groups.perm_gps.permgroup_named.**PSL**(*n*, *q*, *name='a'*)

Bases: *PermutationGroup_plg*

The projective special linear groups over $\mathbf{F}_(q)$.

INPUT:

- n – positive integer; the degree
- q – either a prime power (the size of the ground field) or a finite field
- name – (default: 'a') variable name of indeterminate of finite field $\mathbf{F}_(q)$

OUTPUT: the group $\mathrm{PSL}(n, q)$

Note

This group is also available via groups.permutation.PSL().

EXAMPLES:

```
sage: G = PSL(2,3); G
Permutation Group with generators [(2,3,4), (1,2)(3,4)]
sage: G.order()
12
sage: G.base_ring()
Finite Field of size 3
sage: print(G)
The projective special linear group of degree 2 over Finite Field of size 3
```

```
>>> from sage.all import *
>>> G = PSL(Integer(2),Integer(3)); G
Permutation Group with generators [(2,3,4), (1,2)(3,4)]
>>> G.order()
12
>>> G.base_ring()
Finite Field of size 3
>>> print(G)
The projective special linear group of degree 2 over Finite Field of size 3
```

We create two groups over nontrivial finite fields:

```
sage: G = PSL(2, 4, 'b'); G                                          #␣
↪needs sage.rings.finite_rings
Permutation Group with generators [(3,4,5), (1,2,3)]
sage: G.base_ring()
Finite Field in b of size 2^2
sage: G = PSL(2, 8); G                                               #␣
↪needs sage.rings.finite_rings
Permutation Group with generators [(3,8,6,4,9,7,5), (1,2,3)(4,7,5)(6,9,8)]
sage: G.base_ring()
```

(continues on next page)

(continued from previous page)

```
Finite Field in a of size 2^3

sage: G.category()
Category of finite enumerated permutation groups
sage: TestSuite(G).run()   # long time
```

```
>>> from sage.all import *
>>> G = PSL(Integer(2), Integer(4), 'b'); G                                   ␣
↪          # needs sage.rings.finite_rings
Permutation Group with generators [(3,4,5), (1,2,3)]
>>> G.base_ring()
Finite Field in b of size 2^2
>>> G = PSL(Integer(2), Integer(8)); G                                        ␣
↪          # needs sage.rings.finite_rings
Permutation Group with generators [(3,8,6,4,9,7,5), (1,2,3)(4,7,5)(6,9,8)]
>>> G.base_ring()
Finite Field in a of size 2^3

>>> G.category()
Category of finite enumerated permutation groups
>>> TestSuite(G).run()   # long time
```

ramification_module_decomposition_hurwitz_curve()

Helps compute the decomposition of the ramification module for the Hurwitz curves X (over **C** say) with automorphism group G = PSL(2,q), q a "Hurwitz prime" (ie, p is $\pm 1 \pmod 7$). Using this computation and Borne's formula helps determine the G-module structure of the RR spaces of equivariant divisors can be determined explicitly.

The output is a list of integer multiplicities: [m1,…,mn], where n is the number of conj classes of G=PSL(2,p) and mi is the multiplicity of pi_i in the ramification module of a Hurwitz curve with automorphism group G. Here IrrRepns(G) = [pi_1,…,pi_n] (in the order listed in the output of self.character_table()).

REFERENCE:

David Joyner, Amy Ksir, Roger Vogeler, "Group representations on Riemann-Roch spaces of some Hurwitz curves," preprint, 2006.

EXAMPLES:

```
sage: G = PSL(2,13)
sage: G.ramification_module_decomposition_hurwitz_curve()   # random, optional␣
↪- gap_packages
[0, 7, 7, 12, 12, 12, 13, 15, 14]
```

```
>>> from sage.all import *
>>> G = PSL(Integer(2),Integer(13))
>>> G.ramification_module_decomposition_hurwitz_curve()   # random, optional -␣
↪gap_packages
[0, 7, 7, 12, 12, 12, 13, 15, 14]
```

This means, for example, that the trivial representation does not occur in the ramification module of a Hurwitz curve with automorphism group PSL(2,13), since the trivial representation is listed first and that entry has multiplicity 0. The "randomness" is due to the fact that GAP randomly orders the conjugacy classes of the same order in the list of all conjugacy classes. Similarly, there is some randomness to the ordering of the characters.

If you try to use this function on a group PSL(2,q) where q is not a (smallish) "Hurwitz prime", an error message will be printed.

ramification_module_decomposition_modular_curve()

Helps compute the decomposition of the ramification module for the modular curve X(p) (over CC say) with automorphism group G = PSL(2,q), q a prime > 5. Using this computation and Borne's formula helps determine the G-module structure of the RR spaces of equivariant divisors can be determined explicitly.

The output is a list of integer multiplicities: [m1,…,mn], where n is the number of conj classes of G=PSL(2,p) and mi is the multiplicity of pi_i in the ramification module of a modular curve with automorphism group G. Here IrrRepns(G) = [pi_1,…,pi_n] (in the order listed in the output of self.character_table()).

REFERENCE: D. Joyner and A. Ksir, 'Modular representations

on some Riemann-Roch spaces of modular curves $X(N)$', Computational Aspects of Algebraic Curves, (Editor: T. Shaska) Lecture Notes in Computing, WorldScientific, 2005.)

EXAMPLES:

```
sage: G = PSL(2,7)
sage: G.ramification_module_decomposition_modular_curve()  # random, optional␣
↪- gap_packages
[0, 4, 3, 6, 7, 8]
```

```
>>> from sage.all import *
>>> G = PSL(Integer(2),Integer(7))
>>> G.ramification_module_decomposition_modular_curve()  # random, optional -␣
↪gap_packages
[0, 4, 3, 6, 7, 8]
```

This means, for example, that the trivial representation does not occur in the ramification module of X(7), since the trivial representation is listed first and that entry has multiplicity 0. The "randomness" is due to the fact that GAP randomly orders the conjugacy classes of the same order in the list of all conjugacy classes. Similarly, there is some randomness to the ordering of the characters.

sage.groups.perm_gps.permgroup_named.**PSP**

alias of `PSp`

class sage.groups.perm_gps.permgroup_named.**PSU**(*n*, *q*, *name='a'*)

Bases: `PermutationGroup_pug`

The projective special unitary groups over $\mathbf{F}_(q)$.

INPUT:

- `n` – positive integer; the degree
- `q` – prime power; the size of the ground field
- `name` – (default: `'a'`) variable name of indeterminate of finite field $\mathbf{F}_(q)$

OUTPUT: PSU(n,`q`)

> **Note**
>
> This group is also available via `groups.permutation.PSU()`.

EXAMPLES:

```
sage: PSU(2,3)                                                              #␣
↪needs sage.rings.finite_rings
The projective special unitary group of degree 2 over Finite Field of size 3

sage: G = PSU(2, 8, name='alpha'); G                                        #␣
↪needs sage.rings.finite_rings
The projective special unitary group of degree 2 over Finite Field in alpha of␣
↪size 2^3
sage: G.base_ring()                                                         #␣
↪needs sage.rings.finite_rings
Finite Field in alpha of size 2^3
```

```
>>> from sage.all import *
>>> PSU(Integer(2),Integer(3))                                                ␣
↪         # needs sage.rings.finite_rings
The projective special unitary group of degree 2 over Finite Field of size 3

>>> G = PSU(Integer(2), Integer(8), name='alpha'); G                          ␣
↪         # needs sage.rings.finite_rings
The projective special unitary group of degree 2 over Finite Field in alpha of␣
↪size 2^3
>>> G.base_ring()                                                    # needs␣
↪sage.rings.finite_rings
Finite Field in alpha of size 2^3
```

class sage.groups.perm_gps.permgroup_named.**PSp**(*n*, *q*, *name='a'*)

Bases: *PermutationGroup_plg*

The projective symplectic linear groups over $\mathbf{F}_(q)$.

INPUT:

- n – positive integer; the degree
- q – prime power; the size of the ground field
- name – (default: 'a') variable name of indeterminate of finite field $\mathbf{F}_(q)$

OUTPUT: PSp(n, q)

Note

This group is also available via groups.permutation.PSp().

EXAMPLES:

```
sage: G = PSp(2,3); G
Permutation Group with generators [(2,3,4), (1,2)(3,4)]
sage: G.order()
12
sage: G = PSp(4,3); G
Permutation Group with generators [(3,4)(6,7)(9,10)(12,13)(17,20)(18,21)(19,
↪22)(23,32)(24,33)(25,34)(26,38)(27,39)(28,40)(29,35)(30,36)(31,37), (1,5,14,17,
```

(continues on next page)

(continued from previous page)

```
↪27,22,19,36,3)(2,6,32)(4,7,23,20,37,13,16,26,40)(8,24,29,30,39,10,33,11,34)(9,
↪15,35)(12,25,38)(21,28,31)]
sage: G.order()
25920
sage: print(G)
The projective symplectic linear group of degree 4 over Finite Field of size 3
sage: G.base_ring()
Finite Field of size 3

sage: G = PSp(2, 8, name='alpha'); G                                    #␣
↪needs sage.rings.finite_rings
Permutation Group with generators [(3,8,6,4,9,7,5), (1,2,3)(4,7,5)(6,9,8)]
sage: G.base_ring()
Finite Field in alpha of size 2^3
```

```
>>> from sage.all import *
>>> G = PSp(Integer(2),Integer(3)); G
Permutation Group with generators [(2,3,4), (1,2)(3,4)]
>>> G.order()
12
>>> G = PSp(Integer(4),Integer(3)); G
Permutation Group with generators [(3,4)(6,7)(9,10)(12,13)(17,20)(18,21)(19,
↪22)(23,32)(24,33)(25,34)(26,38)(27,39)(28,40)(29,35)(30,36)(31,37), (1,5,14,17,
↪27,22,19,36,3)(2,6,32)(4,7,23,20,37,13,16,26,40)(8,24,29,30,39,10,33,11,34)(9,
↪15,35)(12,25,38)(21,28,31)]
>>> G.order()
25920
>>> print(G)
The projective symplectic linear group of degree 4 over Finite Field of size 3
>>> G.base_ring()
Finite Field of size 3

>>> G = PSp(Integer(2), Integer(8), name='alpha'); G                          ␣
↪           # needs sage.rings.finite_rings
Permutation Group with generators [(3,8,6,4,9,7,5), (1,2,3)(4,7,5)(6,9,8)]
>>> G.base_ring()
Finite Field in alpha of size 2^3
```

class sage.groups.perm_gps.permgroup_named.**PermutationGroup_plg**(*gens=None*, *gap_group=None*, *canonicalize=True*, *domain=None*, *category=None*)

Bases: *PermutationGroup_unique*

base_ring()

EXAMPLES:

```
sage: G = PGL(2,3)
sage: G.base_ring()
Finite Field of size 3

sage: G = PSL(2,3)
```

(continues on next page)

(continued from previous page)

```
sage: G.base_ring()
Finite Field of size 3
```

```
>>> from sage.all import *
>>> G = PGL(Integer(2),Integer(3))
>>> G.base_ring()
Finite Field of size 3

>>> G = PSL(Integer(2),Integer(3))
>>> G.base_ring()
Finite Field of size 3
```

matrix_degree()

EXAMPLES:

```
sage: G = PSL(2,3)
sage: G.matrix_degree()
2
```

```
>>> from sage.all import *
>>> G = PSL(Integer(2),Integer(3))
>>> G.matrix_degree()
2
```

class sage.groups.perm_gps.permgroup_named.**PermutationGroup_pug**(*gens=None*, *gap_group=None*, *canonicalize=True*, *domain=None*, *category=None*)

Bases: *PermutationGroup_plg*

field_of_definition()

EXAMPLES:

```
sage: PSU(2,3).field_of_definition()                                        #␣
↪needs sage.rings.finite_rings
Finite Field in a of size 3^2
```

```
>>> from sage.all import *
>>> PSU(Integer(2),Integer(3)).field_of_definition()                        ␣
↪                # needs sage.rings.finite_rings
Finite Field in a of size 3^2
```

class sage.groups.perm_gps.permgroup_named.**PermutationGroup_symalt**(*gens=None*, *gap_group=None*, *canonicalize=True*, *domain=None*, *category=None*)

Bases: *PermutationGroup_unique*

This is a class used to factor out some of the commonality in the *SymmetricGroup* and *AlternatingGroup* classes.

class sage.groups.perm_gps.permgroup_named.**PermutationGroup_unique**(*gens=None*, *gap_group=None*, *canonicalize=True*, *domain=None*, *category=None*)

Bases: CachedRepresentation, *PermutationGroup_generic*

Todo

Fix the broken hash.

```
sage: G = SymmetricGroup(6)
sage: G3 = G.subgroup([G((1,2,3,4,5,6)),G((1,2))])
sage: hash(G) == hash(G3)  # todo: Should be True!
False
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(6))
>>> G3 = G.subgroup([G((Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6))),G((Integer(1),Integer(2)))])
>>> hash(G) == hash(G3)  # todo: Should be True!
False
```

class sage.groups.perm_gps.permgroup_named.**PrimitiveGroup**(*d*, *n*)

Bases: *PermutationGroup_unique*

The primitive group from the GAP tables of primitive groups.

INPUT:

- d – nonnegative integer; the degree of the group
- n – positive integer; the index of the group in the GAP database, starting at 1

OUTPUT: the n-th primitive group of degree d

EXAMPLES:

```
sage: PrimitiveGroup(0,1)
Trivial group
sage: PrimitiveGroup(1,1)
Trivial group
sage: G = PrimitiveGroup(5, 2); G
D(2*5)
sage: G.gens()
((2,4)(3,5), (1,2,3,5,4))
sage: G.category()
Category of finite enumerated permutation groups
```

```
>>> from sage.all import *
>>> PrimitiveGroup(Integer(0),Integer(1))
Trivial group
>>> PrimitiveGroup(Integer(1),Integer(1))
Trivial group
>>> G = PrimitiveGroup(Integer(5), Integer(2)); G
```

(continues on next page)

(continued from previous page)

```
D(2*5)
>>> G.gens()
((2,4)(3,5), (1,2,3,5,4))
>>> G.category()
Category of finite enumerated permutation groups
```

Warning

this follows GAP's naming convention of indexing the primitive groups starting from `1`:

```
sage: PrimitiveGroup(5,0)
Traceback (most recent call last):
...
ValueError: index n must be in {1,..,5}
```

```
>>> from sage.all import *
>>> PrimitiveGroup(Integer(5),Integer(0))
Traceback (most recent call last):
...
ValueError: index n must be in {1,..,5}
```

Only primitive groups of "small" degree are available in GAP's database:

```
sage: PrimitiveGroup(2^12,1)
Traceback (most recent call last):
...
GAPError: Error, Primitive groups of degree 4096 are not known!
```

```
>>> from sage.all import *
>>> PrimitiveGroup(Integer(2)**Integer(12),Integer(1))
Traceback (most recent call last):
...
GAPError: Error, Primitive groups of degree 4096 are not known!
```

group_primitive_id()

Return the index of this group in the GAP database of primitive groups.

OUTPUT: a positive integer, following GAP's conventions

EXAMPLES:

```
sage: G = PrimitiveGroup(5,2); G.group_primitive_id()
2
```

```
>>> from sage.all import *
>>> G = PrimitiveGroup(Integer(5),Integer(2)); G.group_primitive_id()
2
```

sage.groups.perm_gps.permgroup_named.**PrimitiveGroups**(*d=None*)

Return the set of all primitive groups of a given degree `d`.

INPUT:

- d – integer (optional)

OUTPUT:

The set of all primitive groups of a given degree d up to isomorphisms using GAP. If d is not specified, it returns the set of all primitive groups up to isomorphisms stored in GAP.

EXAMPLES:

```
sage: PrimitiveGroups(3)
Primitive Groups of degree 3
sage: PrimitiveGroups(7)
Primitive Groups of degree 7
sage: PrimitiveGroups(8)
Primitive Groups of degree 8
sage: PrimitiveGroups()
Primitive Groups
```

```
>>> from sage.all import *
>>> PrimitiveGroups(Integer(3))
Primitive Groups of degree 3
>>> PrimitiveGroups(Integer(7))
Primitive Groups of degree 7
>>> PrimitiveGroups(Integer(8))
Primitive Groups of degree 8
>>> PrimitiveGroups()
Primitive Groups
```

The database is currently limited:

```
sage: PrimitiveGroups(2^12).cardinality()
Traceback (most recent call last):
...
GAPError: Error, Primitive groups of degree 4096 are not known!
```

```
>>> from sage.all import *
>>> PrimitiveGroups(Integer(2)**Integer(12)).cardinality()
Traceback (most recent call last):
...
GAPError: Error, Primitive groups of degree 4096 are not known!
```

Todo

This enumeration helper could be extended based on PrimitiveGroupsIterator in GAP. This method allows to enumerate groups with specified properties such as transitivity, solvability, …, without creating all groups.

class sage.groups.perm_gps.permgroup_named.**PrimitiveGroupsAll**

Bases: DisjointUnionEnumeratedSets

The infinite set of all primitive groups up to isomorphisms.

EXAMPLES:

```
sage: L = PrimitiveGroups(); L
Primitive Groups
sage: L.category()
Category of facade infinite enumerated sets
sage: L.cardinality()
+Infinity

sage: p = L.__iter__()
sage: (next(p), next(p), next(p), next(p),
....:  next(p), next(p), next(p), next(p))
(Trivial group, Trivial group, S(2), A(3), S(3), A(4), S(4), C(5))
```

```
>>> from sage.all import *
>>> L = PrimitiveGroups(); L
Primitive Groups
>>> L.category()
Category of facade infinite enumerated sets
>>> L.cardinality()
+Infinity

>>> p = L.__iter__()
>>> (next(p), next(p), next(p), next(p),
...  next(p), next(p), next(p), next(p))
(Trivial group, Trivial group, S(2), A(3), S(3), A(4), S(4), C(5))
```

class sage.groups.perm_gps.permgroup_named.**PrimitiveGroupsOfDegree**(*n*)

Bases: CachedRepresentation, Parent

The set of all primitive groups of a given degree up to isomorphisms.

EXAMPLES:

```
sage: S = PrimitiveGroups(5); S
Primitive Groups of degree 5
sage: S.list()
[C(5), D(2*5), AGL(1, 5), A(5), S(5)]
sage: S.an_element()
C(5)
```

```
>>> from sage.all import *
>>> S = PrimitiveGroups(Integer(5)); S
Primitive Groups of degree 5
>>> S.list()
[C(5), D(2*5), AGL(1, 5), A(5), S(5)]
>>> S.an_element()
C(5)
```

We write the cardinality of all primitive groups of degree 5:

```
sage: for G in PrimitiveGroups(5):
....:     print(G.cardinality())
5
10
```

(continues on next page)

(continued from previous page)

```
20
60
120
```

```
>>> from sage.all import *
>>> for G in PrimitiveGroups(Integer(5)):
...     print(G.cardinality())
5
10
20
60
120
```

cardinality()

Return the cardinality of `self`.

OUTPUT: integer; the number of primitive groups of a given degree up to isomorphism

EXAMPLES:

```
sage: PrimitiveGroups(0).cardinality()
1
sage: PrimitiveGroups(2).cardinality()
1
sage: PrimitiveGroups(7).cardinality()
7
sage: PrimitiveGroups(12).cardinality()
6
sage: [PrimitiveGroups(i).cardinality() for i in range(11)]
[1, 1, 1, 2, 2, 5, 4, 7, 7, 11, 9]
```

```
>>> from sage.all import *
>>> PrimitiveGroups(Integer(0)).cardinality()
1
>>> PrimitiveGroups(Integer(2)).cardinality()
1
>>> PrimitiveGroups(Integer(7)).cardinality()
7
>>> PrimitiveGroups(Integer(12)).cardinality()
6
>>> [PrimitiveGroups(i).cardinality() for i in range(Integer(11))]
[1, 1, 1, 2, 2, 5, 4, 7, 7, 11, 9]
```

class `sage.groups.perm_gps.permgroup_named.`**QuaternionGroup**

Bases: *DiCyclicGroup*

The quaternion group of order 8.

OUTPUT:

The quaternion group of order 8, as a permutation group. See the *DiCyclicGroup* class for a generalization of this construction.

Note: This group is also available via `groups.permutation.Quaternion()`.

EXAMPLES:

The quaternion group is one of two non-abelian groups of order 8, the other being the dihedral group D_4. One way to describe this group is with three generators, I, J, K, so the whole group is then given as the set $\{\pm 1, \pm I, \pm J, \pm K\}$ with relations such as $I^2 = J^2 = K^2 = -1$, $IJ = K$ and $JI = -K$.

The examples below illustrate how to use this group in a similar manner, by testing some of these relations. The representation used here is the left-regular representation.

```
sage: Q = QuaternionGroup()
sage: I = Q.gen(0)
sage: J = Q.gen(1)
sage: K = I*J
sage: [I,J,K]
[(1,2,3,4)(5,6,7,8), (1,5,3,7)(2,8,4,6), (1,8,3,6)(2,7,4,5)]
sage: neg_one = I^2; neg_one
(1,3)(2,4)(5,7)(6,8)
sage: J^2 == neg_one and K^2 == neg_one
True
sage: J*I == neg_one*K
True
sage: Q.center().order() == 2
True
sage: neg_one in Q.center()
True
```

```
>>> from sage.all import *
>>> Q = QuaternionGroup()
>>> I = Q.gen(Integer(0))
>>> J = Q.gen(Integer(1))
>>> K = I*J
>>> [I,J,K]
[(1,2,3,4)(5,6,7,8), (1,5,3,7)(2,8,4,6), (1,8,3,6)(2,7,4,5)]
>>> neg_one = I**Integer(2); neg_one
(1,3)(2,4)(5,7)(6,8)
>>> J**Integer(2) == neg_one and K**Integer(2) == neg_one
True
>>> J*I == neg_one*K
True
>>> Q.center().order() == Integer(2)
True
>>> neg_one in Q.center()
True
```

AUTHOR:

- Rob Beezer (2009-10-09)

class `sage.groups.perm_gps.permgroup_named.`**`SemidihedralGroup`**`(m)`

Bases: *`PermutationGroup_unique`*

The semidihedral group of order 2^m.

INPUT:

- `m` – positive integer; the power of 2 that is the group's order

OUTPUT:

The semidihedral group of order 2^m. These groups can be thought of as a semidirect product of $C_{2^{m-1}}$ with C_2, where the nontrivial element of C_2 is sent to the element of the automorphism group of $C_{2^{m-1}}$ that sends elements to their $-1 + 2^{m-2}$ th power. Thus, the group has the presentation:

$$\langle x, y \mid x^{2^{m-1}}, y^2, y^{-1}xy = x^{-1+2^{m-2}} \rangle$$

This family is notable because it is made up of non-abelian 2-groups that all contain cyclic subgroups of index 2. It is one of only four such families.

EXAMPLES:

In [Gor1980] it is shown that the semidihedral groups have center of order 2. It is also shown that they have a Frattini subgroup equal to their commutator, which is a cyclic subgroup of order 2^{m-2}.

```
sage: G = SemidihedralGroup(12)
sage: G.order() == 2^12
True
sage: G.commutator() == G.frattini_subgroup()
True
sage: G.commutator().order() == 2^10
True
sage: G.commutator().is_cyclic()
True
sage: G.center().order()
2

sage: G = SemidihedralGroup(4)
sage: len([H for H in G.subgroups() if H.is_cyclic() and H.order() == 8])
1
sage: G.gens()
((2,4)(3,7)(6,8), (1,2,3,4,5,6,7,8))
sage: x = G.gens()[1]; y = G.gens()[0]
sage: x.order() == 2^3; y.order() == 2
True
True
sage: y*x*y == x^(-1+2^2)
True
```

```
>>> from sage.all import *
>>> G = SemidihedralGroup(Integer(12))
>>> G.order() == Integer(2)**Integer(12)
True
>>> G.commutator() == G.frattini_subgroup()
True
>>> G.commutator().order() == Integer(2)**Integer(10)
True
>>> G.commutator().is_cyclic()
True
```

(continues on next page)

(continued from previous page)

```
>>> G.center().order()
2

>>> G = SemidihedralGroup(Integer(4))
>>> len([H for H in G.subgroups() if H.is_cyclic() and H.order() == Integer(8)])
1
>>> G.gens()
((2,4)(3,7)(6,8), (1,2,3,4,5,6,7,8))
>>> x = G.gens()[Integer(1)]; y = G.gens()[Integer(0)]
>>> x.order() == Integer(2)**Integer(3); y.order() == Integer(2)
True
True
>>> y*x*y == x**(-Integer(1)+Integer(2)**Integer(2))
True
```

AUTHOR:

- Kevin Halasz (2012-8-7)

class sage.groups.perm_gps.permgroup_named.**SmallPermutationGroup**(*order*, *gap_id*)

Bases: *PermutationGroup_generic*

A GAP SmallGroup, returned as a permutation group.

GAP contains a library SGL of small groups, each identified by its GAP SmallGroup id. (MAGMA uses the same identifiers). The GAP SmallGroup id is a pair `[n,k]` consisting of `n`, the order of the group, and `k`, an index determining the group specifically. This class can construct the group as a permutation group from this data.

INPUT:

- `order` – the order of the group
- `gap_id` – the numerical index in the GAP id of the group

Generators may be obtained through the `gens()` method. These could change for a particular group in later releases of GAP. In many instances the degree of the constructed group `SmallPermutationGroup(n,k)` will be a permutation group on n letters, but this will not always be true.

EXAMPLES:

```
sage: G = SmallPermutationGroup(12,4); G
Group of order 12 and GAP Id 4 as a permutation group
sage: G.gens()
((4,5), (1,2), (3,4,5))
sage: G.character_table()                                                     #␣
↪needs sage.rings.number_field
[ 1  1  1  1  1  1]
[ 1 -1  1 -1  1 -1]
[ 1 -1  1  1 -1  1]
[ 1  1  1 -1 -1 -1]
[ 2  0 -1 -2  0  1]
[ 2  0 -1  2  0 -1]
sage: def numgps(n): return ZZ(libgap.NumberSmallGroups(n))
sage: all(SmallPermutationGroup(n,k).id() == [n,k]
....:     for n in [1..64] for k in [1..numgps(n)])   # long time (180s)
True
```

(continues on next page)

(continued from previous page)

```
sage: H = SmallPermutationGroup(6,1)
sage: H.is_abelian()
False
sage: [H.centralizer(g) for g in H.conjugacy_classes_representatives()]
[Subgroup generated by [(1,3), (2,3)] of
  (Group of order 6 and GAP Id 1 as a permutation group),
 Subgroup generated by [(2,3)] of
  (Group of order 6 and GAP Id 1 as a permutation group),
 Subgroup generated by [(1,2,3)] of
  (Group of order 6 and GAP Id 1 as a permutation group)]
```

```
>>> from sage.all import *
>>> G = SmallPermutationGroup(Integer(12),Integer(4)); G
Group of order 12 and GAP Id 4 as a permutation group
>>> G.gens()
((4,5), (1,2), (3,4,5))
>>> G.character_table()                                                   #␣
↪needs sage.rings.number_field
[ 1  1  1  1  1  1]
[ 1 -1  1 -1  1 -1]
[ 1 -1  1  1 -1  1]
[ 1  1  1 -1 -1 -1]
[ 2  0 -1 -2  0  1]
[ 2  0 -1  2  0 -1]
>>> def numgps(n): return ZZ(libgap.NumberSmallGroups(n))
>>> all(SmallPermutationGroup(n,k).id() == [n,k]
...     for n in (ellipsis_range(Integer(1),Ellipsis,Integer(64))) for k in␣
↪(ellipsis_range(Integer(1),Ellipsis,numgps(n))))   # long time (180s)
True
>>> H = SmallPermutationGroup(Integer(6),Integer(1))
>>> H.is_abelian()
False
>>> [H.centralizer(g) for g in H.conjugacy_classes_representatives()]
[Subgroup generated by [(1,3), (2,3)] of
  (Group of order 6 and GAP Id 1 as a permutation group),
 Subgroup generated by [(2,3)] of
  (Group of order 6 and GAP Id 1 as a permutation group),
 Subgroup generated by [(1,2,3)] of
  (Group of order 6 and GAP Id 1 as a permutation group)]
```

gap_small_group()

Return the GAP small group object corresponding to `self`.

GAP realizes some small groups as `PermutationGroup`, others as `PcGroups` (polycyclic groups). The *SmallPermutationGroup* class always returns a `PermutationGroup`, but in the process of creating this group a GAP `SmallGroup` is generated. This method returns that group.

EXAMPLES:

```
sage: SmallPermutationGroup(168,41).gap_small_group()
<pc group of size 168 with 5 generators>
sage: SmallPermutationGroup(168,42).gap_small_group()
Group([ (3,4)(5,6), (1,2,3)(4,5,7) ])
```

```
>>> from sage.all import *
>>> SmallPermutationGroup(Integer(168),Integer(41)).gap_small_group()
<pc group of size 168 with 5 generators>
>>> SmallPermutationGroup(Integer(168),Integer(42)).gap_small_group()
Group([ (3,4)(5,6), (1,2,3)(4,5,7) ])
```

order()

Return the order of the group corresponding to `self`.

EXAMPLES:

```
sage: [SmallPermutationGroup(21,k).order() for k in [1,2]]
[21, 21]
```

```
>>> from sage.all import *
>>> [SmallPermutationGroup(Integer(21),k).order() for k in [Integer(1),
↪Integer(2)]]
[21, 21]
```

class sage.groups.perm_gps.permgroup_named.**SplitMetacyclicGroup**(*p*, *m*)

Bases: *PermutationGroup_unique*

The split metacyclic group of order p^m.

INPUT:

- `p` – a prime number that is the prime underlying this p-group
- `m` – positive integer such that the order of this group is the p^m. Be aware that, for even p, m must be greater than 3, while for odd p, m must be greater than 2.

OUTPUT:

The split metacyclic group of order p^m. This family of groups has presentation

$$\langle x, y \mid x^{p^{m-1}}, y^p, y^{-1}xy = x^{1+p^{m-2}} \rangle$$

This family is notable because, for odd p, these are the only p-groups with a cyclic subgroup of index p, a result proven in [Gor1980]. It is also shown in [Gor1980] that this is one of four families containing nonabelian 2-groups with a cyclic subgroup of index 2 (with the others being the dicyclic groups, the dihedral groups, and the semidihedral groups).

EXAMPLES:

Using the last relation in the group's presentation, one can see that the elements of the form $y^i x$, $0 \leq i \leq p-1$ all have order p^{m-1}, as it can be shown that their p th powers are all $x^{p^{m-2}+p}$, an element with order p^{m-2}. Manipulation of the same relation shows that none of these elements are powers of any other. Thus, there are p cyclic maximal subgroups in each split metacyclic group. It is also proven in [Gor1980] that this family has commutator subgroup of order p, and the Frattini subgroup is equal to the center, with this group being cyclic of order p^{m-2}. These characteristics are necessary to identify these groups in the case that $p = 2$, although the possession of a cyclic maximal subgroup in a non-abelian p-group is enough for odd p given the group's order.

```
sage: G = SplitMetacyclicGroup(2,8)
sage: G.order() == 2**8
True
sage: G.is_abelian()
False
```

(continues on next page)

(continued from previous page)

```
sage: len([H for H in G.subgroups() if H.order() == 2^7 and H.is_cyclic()])
2
sage: G.commutator().order()
2
sage: G.frattini_subgroup() == G.center()
True
sage: G.center().order() == 2^6
True
sage: G.center().is_cyclic()
True

sage: G = SplitMetacyclicGroup(3,3)
sage: len([H for H in G.subgroups() if H.order() == 3^2 and H.is_cyclic()])
3
sage: G.commutator().order()
3
sage: G.frattini_subgroup() == G.center()
True
sage: G.center().order()
3
```

```
>>> from sage.all import *
>>> G = SplitMetacyclicGroup(Integer(2),Integer(8))
>>> G.order() == Integer(2)**Integer(8)
True
>>> G.is_abelian()
False
>>> len([H for H in G.subgroups() if H.order() == Integer(2)**Integer(7) and H.is_
↪cyclic()])
2
>>> G.commutator().order()
2
>>> G.frattini_subgroup() == G.center()
True
>>> G.center().order() == Integer(2)**Integer(6)
True
>>> G.center().is_cyclic()
True

>>> G = SplitMetacyclicGroup(Integer(3),Integer(3))
>>> len([H for H in G.subgroups() if H.order() == Integer(3)**Integer(2) and H.is_
↪cyclic()])
3
>>> G.commutator().order()
3
>>> G.frattini_subgroup() == G.center()
True
>>> G.center().order()
3
```

AUTHOR:

- Kevin Halasz (2012-8-7)

class sage.groups.perm_gps.permgroup_named.**SuzukiGroup**(*q*, *name='a'*)

Bases: *PermutationGroup_unique*

The Suzuki group over $\mathbf{F}_(q)$, ${}^2B_2(2^{2k+1}) = Sz(2^{2k+1})$.

A wrapper for the GAP function `SuzukiGroup`.

INPUT:

- `q` – 2^n, an odd power of 2; the size of the ground field. (Strictly speaking, n should be greater than 1, or else this group is not simple.)
- `name` – (default: `'a'`) variable name of indeterminate of finite field $\mathbf{F}_(q)$

OUTPUT: a Suzuki group

Note

This group is also available via `groups.permutation.Suzuki()`.

EXAMPLES:

```
sage: SuzukiGroup(8)                                                    #␣
↪needs sage.rings.finite_rings
Permutation Group with generators [(1,28,10,44)(3,50,11,42)(4,43,53,64)(5,9,39,
↪52)(6,36,63,13)(7,51,60,57)(8,33,37,16)(12,24,55,29)(14,30,48,47)(15,19,61,
↪54)(17,59,22,62)(18,23,34,31)(20,38,49,25)(21,26,45,58)(27,32,41,65)(35,46,40,
↪56),
 (1,2)(3,10)(4,42)(5,18)(6,50)(7,26)(8,58)(9,34)(12,28)(13,45)(14,44)(15,23)(16,
↪31)(17,21)(19,39)(20,38)(22,25)(24,61)(27,60)(29,65)(30,55)(32,33)(35,52)(36,
↪49)(37,59)(40,54)(41,62)(43,53)(46,48)(47,56)(51,63)(57,64)]
sage: print(SuzukiGroup(8))                                             #␣
↪needs sage.rings.finite_rings
The Suzuki group over Finite Field in a of size 2^3

sage: # needs sage.rings.finite_rings
sage: G = SuzukiGroup(32, name='alpha')
sage: G.order()
32537600
sage: G.order().factor()
2^10 * 5^2 * 31 * 41
sage: G.base_ring()
Finite Field in alpha of size 2^5
```

```
>>> from sage.all import *
>>> SuzukiGroup(Integer(8))                                                   ␣
↪ # needs sage.rings.finite_rings
Permutation Group with generators [(1,28,10,44)(3,50,11,42)(4,43,53,64)(5,9,39,
↪52)(6,36,63,13)(7,51,60,57)(8,33,37,16)(12,24,55,29)(14,30,48,47)(15,19,61,
↪54)(17,59,22,62)(18,23,34,31)(20,38,49,25)(21,26,45,58)(27,32,41,65)(35,46,40,
↪56),
 (1,2)(3,10)(4,42)(5,18)(6,50)(7,26)(8,58)(9,34)(12,28)(13,45)(14,44)(15,23)(16,
↪31)(17,21)(19,39)(20,38)(22,25)(24,61)(27,60)(29,65)(30,55)(32,33)(35,52)(36,
↪49)(37,59)(40,54)(41,62)(43,53)(46,48)(47,56)(51,63)(57,64)]
>>> print(SuzukiGroup(Integer(8)))                                            ␣
```

(continues on next page)

(continued from previous page)

```
↪ # needs sage.rings.finite_rings
The Suzuki group over Finite Field in a of size 2^3

>>> # needs sage.rings.finite_rings
>>> G = SuzukiGroup(Integer(32), name='alpha')
>>> G.order()
32537600
>>> G.order().factor()
2^10 * 5^2 * 31 * 41
>>> G.base_ring()
Finite Field in alpha of size 2^5
```

REFERENCES:

- Wikipedia article Group_of_Lie_type#Suzuki-Ree_groups

base_ring()

EXAMPLES:

```
sage: G = SuzukiGroup(32, name='alpha')                                 #␣
↪needs sage.rings.finite_rings
sage: G.base_ring()                                                     #␣
↪needs sage.rings.finite_rings
Finite Field in alpha of size 2^5
```

```
>>> from sage.all import *
>>> G = SuzukiGroup(Integer(32), name='alpha')                           ␣
↪     # needs sage.rings.finite_rings
>>> G.base_ring()                                                      #␣
↪needs sage.rings.finite_rings
Finite Field in alpha of size 2^5
```

class sage.groups.perm_gps.permgroup_named.**SuzukiSporadicGroup**

Bases: *PermutationGroup_unique*

Suzuki Sporadic Group.

EXAMPLES:

```
sage: G = groups.permutation.SuzukiSporadic(); G                 # optional -␣
↪gap_package_atlasrep internet
Sporadic Suzuki group acting on 1782 points
```

```
>>> from sage.all import *
>>> G = groups.permutation.SuzukiSporadic(); G                 # optional -␣
↪gap_package_atlasrep internet
Sporadic Suzuki group acting on 1782 points
```

class sage.groups.perm_gps.permgroup_named.**SymmetricGroup**(*domain=None*)

Bases: *PermutationGroup_symalt*

The full symmetric group of order $n!$, as a permutation group.

If n is a list or tuple of positive integers then it returns the symmetric group of the associated set.

INPUT:

- n – positive integer, or list or tuple thereof

Note

This group is also available via `groups.permutation.Symmetric()`.

EXAMPLES:

```
sage: G = SymmetricGroup(8)
sage: G.order()
40320
sage: G
Symmetric group of order 8! as a permutation group
sage: G.degree()
8
sage: S8 = SymmetricGroup(8)
sage: G = SymmetricGroup([1,2,4,5])
sage: G
Symmetric group of order 4! as a permutation group
sage: G.domain()
{1, 2, 4, 5}
sage: G = SymmetricGroup(4)
sage: G
Symmetric group of order 4! as a permutation group
sage: G.domain()
{1, 2, 3, 4}
sage: G.category()
Join of Category of finite enumerated permutation groups and
Category of finite Weyl groups and
Category of well generated finite irreducible complex reflection groups
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(8))
>>> G.order()
40320
>>> G
Symmetric group of order 8! as a permutation group
>>> G.degree()
8
>>> S8 = SymmetricGroup(Integer(8))
>>> G = SymmetricGroup([Integer(1),Integer(2),Integer(4),Integer(5)])
>>> G
Symmetric group of order 4! as a permutation group
>>> G.domain()
{1, 2, 4, 5}
>>> G = SymmetricGroup(Integer(4))
>>> G
Symmetric group of order 4! as a permutation group
>>> G.domain()
{1, 2, 3, 4}
>>> G.category()
Join of Category of finite enumerated permutation groups and
```

(continues on next page)

(continued from previous page)

```
Category of finite Weyl groups and
Category of well generated finite irreducible complex reflection groups
```

Element

alias of *SymmetricGroupElement*

algebra (*base_ring*, *category=None*)

Return the symmetric group algebra associated to `self`.

INPUT:

- `base_ring` – a ring
- `category` – a category (default: the category of `self`)

If `self` is the symmetric group on $1, \ldots, n$, then this is special cased to take advantage of the features in `SymmetricGroupAlgebra`. Otherwise the usual group algebra is returned.

EXAMPLES:

```
sage: S4 = SymmetricGroup(4)
sage: S4.algebra(QQ)                                                    #␣
↪needs sage.combinat
Symmetric group algebra of order 4 over Rational Field

sage: S3 = SymmetricGroup([1,2,3])
sage: A = S3.algebra(QQ); A                                             #␣
↪needs sage.combinat
Symmetric group algebra of order 3 over Rational Field
sage: a = S3.an_element(); a
(2,3)
sage: A(a)                                                              #␣
↪needs sage.combinat
(2,3)
```

```
>>> from sage.all import *
>>> S4 = SymmetricGroup(Integer(4))
>>> S4.algebra(QQ)                                                    #␣
↪needs sage.combinat
Symmetric group algebra of order 4 over Rational Field

>>> S3 = SymmetricGroup([Integer(1),Integer(2),Integer(3)])
>>> A = S3.algebra(QQ); A                                             #␣
↪needs sage.combinat
Symmetric group algebra of order 3 over Rational Field
>>> a = S3.an_element(); a
(2,3)
>>> A(a)                                                              #␣
↪needs sage.combinat
(2,3)
```

We illustrate the choice of the category:

```
sage: A.category()                                                      #␣
↪needs sage.combinat
```

(continues on next page)

(continued from previous page)

```
Join of Category of Coxeter group algebras over Rational Field
    and Category of finite group algebras over Rational Field
    and Category of finite dimensional cellular algebras with basis
         over Rational Field
sage: A = S3.algebra(QQ, category=Semigroups())                              #␣
↪needs sage.combinat
sage: A.category()                                                           #␣
↪needs sage.combinat
Category of finite dimensional unital cellular semigroup algebras
 over Rational Field
```

```
>>> from sage.all import *
>>> A.category()                                                           #␣
↪needs sage.combinat
Join of Category of Coxeter group algebras over Rational Field
    and Category of finite group algebras over Rational Field
    and Category of finite dimensional cellular algebras with basis
         over Rational Field
>>> A = S3.algebra(QQ, category=Semigroups())                              #␣
↪needs sage.combinat
>>> A.category()                                                           #␣
↪needs sage.combinat
Category of finite dimensional unital cellular semigroup algebras
 over Rational Field
```

In the following case, a usual group algebra is returned:

```
sage: S = SymmetricGroup([2,3,5])
sage: S.algebra(QQ)                                                          #␣
↪needs sage.combinat
Algebra of Symmetric group of order 3! as a permutation group over Rational␣
↪Field
sage: a = S.an_element(); a
(3,5)
sage: S.algebra(QQ)(a)                                                       #␣
↪needs sage.combinat
(3,5)
```

```
>>> from sage.all import *
>>> S = SymmetricGroup([Integer(2),Integer(3),Integer(5)])
>>> S.algebra(QQ)                                                          #␣
↪needs sage.combinat
Algebra of Symmetric group of order 3! as a permutation group over Rational␣
↪Field
>>> a = S.an_element(); a
(3,5)
>>> S.algebra(QQ)(a)                                                       #␣
↪needs sage.combinat
(3,5)
```

cartan_type()

Return the Cartan type of `self`.

The symmetric group S_n is a Coxeter group of type A_{n-1}.

EXAMPLES:

```
sage: A = SymmetricGroup([2,3,7]); A.cartan_type()
['A', 2]

sage: A = SymmetricGroup([]); A.cartan_type()
['A', 0]
```

```
>>> from sage.all import *
>>> A = SymmetricGroup([Integer(2),Integer(3),Integer(7)]); A.cartan_type()
['A', 2]

>>> A = SymmetricGroup([]); A.cartan_type()
['A', 0]
```

conjugacy_class(*g*)

Return the conjugacy class of `g` inside the symmetric group `self`.

INPUT:

- `g` – a partition or an element of the symmetric group `self`

OUTPUT: a conjugacy class of a symmetric group

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: g = G((1,2,3,4))
sage: G.conjugacy_class(g)                                                    #␣
↪needs sage.combinat
Conjugacy class of cycle type [4, 1] in
 Symmetric group of order 5! as a permutation group
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> g = G((Integer(1),Integer(2),Integer(3),Integer(4)))
>>> G.conjugacy_class(g)                                                    #␣
↪needs sage.combinat
Conjugacy class of cycle type [4, 1] in
 Symmetric group of order 5! as a permutation group
```

conjugacy_classes()

Return a list of the conjugacy classes of `self`.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.conjugacy_classes()                                                   #␣
↪needs sage.combinat
[Conjugacy class of cycle type [1, 1, 1, 1, 1] in
     Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [2, 1, 1, 1] in
     Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [2, 2, 1] in
```

(continues on next page)

(continued from previous page)

```
    Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [3, 1, 1] in
    Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [3, 2] in
    Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [4, 1] in
    Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [5] in
    Symmetric group of order 5! as a permutation group]
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> G.conjugacy_classes()                                                   #␣
↪needs sage.combinat
[Conjugacy class of cycle type [1, 1, 1, 1, 1] in
    Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [2, 1, 1, 1] in
    Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [2, 2, 1] in
    Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [3, 1, 1] in
    Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [3, 2] in
    Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [4, 1] in
    Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [5] in
    Symmetric group of order 5! as a permutation group]
```

conjugacy_classes_iterator()

Iterate over the conjugacy classes of `self`.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: list(G.conjugacy_classes_iterator()) == G.conjugacy_classes()          #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> list(G.conjugacy_classes_iterator()) == G.conjugacy_classes()            #␣
↪needs sage.combinat
True
```

conjugacy_classes_representatives()

Return a complete list of representatives of conjugacy classes in a permutation group G.

Let S_n be the symmetric group on n letters. The conjugacy classes are indexed by partitions λ of n. The ordering of the conjugacy classes is reverse lexicographic order of the partitions.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.conjugacy_classes_representatives()                                 #␣
↪needs sage.combinat
[(), (1,2), (1,2)(3,4), (1,2,3), (1,2,3)(4,5),
 (1,2,3,4), (1,2,3,4,5)]
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> G.conjugacy_classes_representatives()                                 #␣
↪needs sage.combinat
[(), (1,2), (1,2)(3,4), (1,2,3), (1,2,3)(4,5),
 (1,2,3,4), (1,2,3,4,5)]
```

```
sage: S = SymmetricGroup(['a','b','c'])
sage: S.conjugacy_classes_representatives()                                 #␣
↪needs sage.combinat
[(), ('a','b'), ('a','b','c')]
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(['a','b','c'])
>>> S.conjugacy_classes_representatives()                                 #␣
↪needs sage.combinat
[(), ('a','b'), ('a','b','c')]
```

coxeter_matrix()

Return the Coxeter matrix of `self`.

EXAMPLES:

```
sage: A = SymmetricGroup([2,3,7,'a']); A.coxeter_matrix()                   #␣
↪needs sage.graphs
[1 3 2]
[3 1 3]
[2 3 1]
```

```
>>> from sage.all import *
>>> A = SymmetricGroup([Integer(2),Integer(3),Integer(7),'a']); A.coxeter_
↪matrix()                   # needs sage.graphs
[1 3 2]
[3 1 3]
[2 3 1]
```

index_set()

Return the index set for the descents of the symmetric group `self`.

EXAMPLES:

```
sage: S8 = SymmetricGroup(8)
sage: S8.index_set()
(1, 2, 3, 4, 5, 6, 7)

sage: S = SymmetricGroup([3,1,4,5])
```

(continues on next page)

(continued from previous page)

```
sage: S.index_set()
(3, 1, 4)
```

```
>>> from sage.all import *
>>> S8 = SymmetricGroup(Integer(8))
>>> S8.index_set()
(1, 2, 3, 4, 5, 6, 7)

>>> S = SymmetricGroup([Integer(3),Integer(1),Integer(4),Integer(5)])
>>> S.index_set()
(3, 1, 4)
```

major_index(*parameter=None*)

Return the *major index generating polynomial* of `self`.

This is a gadget counting the elements of `self` by major index.

INPUT:

- `parameter` – an element of a ring; the result is more explicit with a formal variable (default: element `q` of Univariate Polynomial Ring in `q` over Integer Ring)

$$P(q) = \sum_{g \in S_n} q^{\text{major index}(g)}$$

EXAMPLES:

```
sage: S4 = SymmetricGroup(4)
sage: S4.major_index()                                                    #␣
↪needs sage.combinat
q^6 + 3*q^5 + 5*q^4 + 6*q^3 + 5*q^2 + 3*q + 1
sage: K.<t> = QQ[]
sage: S4.major_index(t)                                                   #␣
↪needs sage.combinat
t^6 + 3*t^5 + 5*t^4 + 6*t^3 + 5*t^2 + 3*t + 1
```

```
>>> from sage.all import *
>>> S4 = SymmetricGroup(Integer(4))
>>> S4.major_index()                                                    #␣
↪needs sage.combinat
q^6 + 3*q^5 + 5*q^4 + 6*q^3 + 5*q^2 + 3*q + 1
>>> K = QQ['t']; (t,) = K._first_ngens(1)
>>> S4.major_index(t)                                                   #␣
↪needs sage.combinat
t^6 + 3*t^5 + 5*t^4 + 6*t^3 + 5*t^2 + 3*t + 1
```

reflection(*i*)

Return the i-th reflection of `self`.

For i in $1, \dots, N$, this gives the i-th reflection of `self`.

See also

- `reflections_index_set()`

- *reflections()*

EXAMPLES:

```
sage: S4 = SymmetricGroup(4)
sage: for i in S4.reflection_index_set():
....:     print('%s %s'%(i, S4.reflection(i)))
0 (1,2)
1 (1,3)
2 (1,4)
3 (2,3)
4 (2,4)
5 (3,4)
```

```
>>> from sage.all import *
>>> S4 = SymmetricGroup(Integer(4))
>>> for i in S4.reflection_index_set():
...     print('%s %s'%(i, S4.reflection(i)))
0 (1,2)
1 (1,3)
2 (1,4)
3 (2,3)
4 (2,4)
5 (3,4)
```

reflection_index_set()

Return the index set of the reflections of `self`.

See also

- *reflection()*
- *reflections()*

EXAMPLES:

```
sage: S5 = SymmetricGroup(5)
sage: S5.reflection_index_set()
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
>>> from sage.all import *
>>> S5 = SymmetricGroup(Integer(5))
>>> S5.reflection_index_set()
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

reflections()

Return the list of all reflections in `self`.

EXAMPLES:

```
sage: A = SymmetricGroup(3)
sage: A.reflections()
[(1,2), (1,3), (2,3)]
```

```
>>> from sage.all import *
>>> A = SymmetricGroup(Integer(3))
>>> A.reflections()
[(1,2), (1,3), (2,3)]
```

simple_reflection(*i*)

For i in the index set of `self`, return the elementary transposition $s_i = (i, i + 1)$.

EXAMPLES:

```
sage: A = SymmetricGroup(5)
sage: A.simple_reflection(3)
(3,4)

sage: A = SymmetricGroup([2,3,7])
sage: A.simple_reflections()
Finite family {2: (2,3), 3: (3,7)}
```

```
>>> from sage.all import *
>>> A = SymmetricGroup(Integer(5))
>>> A.simple_reflection(Integer(3))
(3,4)

>>> A = SymmetricGroup([Integer(2),Integer(3),Integer(7)])
>>> A.simple_reflections()
Finite family {2: (2,3), 3: (3,7)}
```

young_subgroup(*comp*)

Return the Young subgroup associated with the composition `comp`.

EXAMPLES:

```
sage: S = SymmetricGroup(8)
sage: c = Composition([2,2,2,2])
sage: S.young_subgroup(c)
Subgroup generated by [(1,2), (3,4), (5,6), (7,8)] of
 (Symmetric group of order 8! as a permutation group)

sage: S = SymmetricGroup(['a','b','c'])
sage: S.young_subgroup([2,1])
Subgroup generated by [('a','b')] of
 (Symmetric group of order 3! as a permutation group)

sage: Y = S.young_subgroup([2,2,2,2,2])
Traceback (most recent call last):
...
ValueError: the composition is not of expected size
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(Integer(8))
>>> c = Composition([Integer(2),Integer(2),Integer(2),Integer(2)])
>>> S.young_subgroup(c)
Subgroup generated by [(1,2), (3,4), (5,6), (7,8)] of
 (Symmetric group of order 8! as a permutation group)

>>> S = SymmetricGroup(['a','b','c'])
>>> S.young_subgroup([Integer(2),Integer(1)])
Subgroup generated by [('a','b')] of
 (Symmetric group of order 3! as a permutation group)

>>> Y = S.young_subgroup([Integer(2),Integer(2),Integer(2),Integer(2),
↪Integer(2)])
Traceback (most recent call last):
...
ValueError: the composition is not of expected size
```

class sage.groups.perm_gps.permgroup_named.**TransitiveGroup**(d, n)

Bases: *PermutationGroup_unique*

The transitive group from the GAP tables of transitive groups.

INPUT:

- d – nonnegative integer; the degree
- n – positive integer; the index of the group in the GAP database, starting at 1

OUTPUT: the n-th transitive group of degree d

Note

This group is also available via groups.permutation.Transitive().

EXAMPLES:

```
sage: TransitiveGroup(0,1)
Transitive group number 1 of degree 0
sage: TransitiveGroup(1,1)
Transitive group number 1 of degree 1
sage: G = TransitiveGroup(5, 2); G
Transitive group number 2 of degree 5
sage: G.gens()
((1,2,3,4,5), (1,4)(2,3))

sage: G.category()
Category of finite enumerated permutation groups
```

```
>>> from sage.all import *
>>> TransitiveGroup(Integer(0),Integer(1))
Transitive group number 1 of degree 0
>>> TransitiveGroup(Integer(1),Integer(1))
Transitive group number 1 of degree 1
```

(continues on next page)

(continued from previous page)

```
>>> G = TransitiveGroup(Integer(5), Integer(2)); G
Transitive group number 2 of degree 5
>>> G.gens()
((1,2,3,4,5), (1,4)(2,3))

>>> G.category()
Category of finite enumerated permutation groups
```

Warning

this follows GAP's naming convention of indexing the transitive groups starting from `1`:

```
sage: TransitiveGroup(5,0)
Traceback (most recent call last):
...
ValueError: index n must be in {1,..,5}
```

```
>>> from sage.all import *
>>> TransitiveGroup(Integer(5),Integer(0))
Traceback (most recent call last):
...
ValueError: index n must be in {1,..,5}
```

Warning

only transitive groups of "small" degree are available in GAP's database:

```
sage: TransitiveGroup(32,1)
Traceback (most recent call last):
...
NotImplementedError: only the transitive groups of degree at most 31
are available in GAP's database
```

```
>>> from sage.all import *
>>> TransitiveGroup(Integer(32),Integer(1))
Traceback (most recent call last):
...
NotImplementedError: only the transitive groups of degree at most 31
are available in GAP's database
```

degree()

Return the degree of this permutation group.

EXAMPLES:

```
sage: TransitiveGroup(8, 44).degree()
8
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> TransitiveGroup(Integer(8), Integer(44)).degree()
8
```

transitive_number()

Return the index of this group in the GAP database, starting at 1.

EXAMPLES:

```
sage: TransitiveGroup(8, 44).transitive_number()
44
```

```
>>> from sage.all import *
>>> TransitiveGroup(Integer(8), Integer(44)).transitive_number()
44
```

sage.groups.perm_gps.permgroup_named.**TransitiveGroups**(*d=None*)

INPUT:

- d – integer (optional)

Return the set of all transitive groups of a given degree d up to isomorphisms. If d is not specified, it returns the set of all transitive groups up to isomorphisms.

EXAMPLES:

```
sage: TransitiveGroups(3)
Transitive Groups of degree 3
sage: TransitiveGroups(7)
Transitive Groups of degree 7
sage: TransitiveGroups(8)
Transitive Groups of degree 8

sage: TransitiveGroups()
Transitive Groups
```

```
>>> from sage.all import *
>>> TransitiveGroups(Integer(3))
Transitive Groups of degree 3
>>> TransitiveGroups(Integer(7))
Transitive Groups of degree 7
>>> TransitiveGroups(Integer(8))
Transitive Groups of degree 8

>>> TransitiveGroups()
Transitive Groups
```

Warning

in practice, the database currently only contains transitive groups up to degree 31:

```
sage: TransitiveGroups(32).cardinality()
Traceback (most recent call last):
...
```

```
NotImplementedError: only the transitive groups of degree at most 31
are available in GAP's database
>>> from sage.all import *
>>> TransitiveGroups(Integer(32)).cardinality()
Traceback (most recent call last):
...
NotImplementedError: only the transitive groups of degree at most 31
are available in GAP's database
```

class sage.groups.perm_gps.permgroup_named.**TransitiveGroupsAll**

Bases: DisjointUnionEnumeratedSets

The infinite set of all transitive groups up to isomorphisms.

EXAMPLES:

```
sage: L = TransitiveGroups(); L
Transitive Groups
sage: L.category()
Category of facade infinite enumerated sets
sage: L.cardinality()
+Infinity

sage: p = L.__iter__()
sage: (next(p), next(p), next(p), next(p), next(p), next(p), next(p), next(p))
(Transitive group number 1 of degree 0, Transitive group number 1 of degree 1,
 Transitive group number 1 of degree 2, Transitive group number 1 of degree 3,
 Transitive group number 2 of degree 3, Transitive group number 1 of degree 4,
 Transitive group number 2 of degree 4, Transitive group number 3 of degree 4)
```

```
>>> from sage.all import *
>>> L = TransitiveGroups(); L
Transitive Groups
>>> L.category()
Category of facade infinite enumerated sets
>>> L.cardinality()
+Infinity

>>> p = L.__iter__()
>>> (next(p), next(p), next(p), next(p), next(p), next(p), next(p), next(p))
(Transitive group number 1 of degree 0, Transitive group number 1 of degree 1,
 Transitive group number 1 of degree 2, Transitive group number 1 of degree 3,
 Transitive group number 2 of degree 3, Transitive group number 1 of degree 4,
 Transitive group number 2 of degree 4, Transitive group number 3 of degree 4)
```

class sage.groups.perm_gps.permgroup_named.**TransitiveGroupsOfDegree**(*n*)

Bases: CachedRepresentation, Parent

The set of all transitive groups of a given (small) degree up to isomorphism.

EXAMPLES:

```
sage: S = TransitiveGroups(4); S
Transitive Groups of degree 4
sage: list(S)
[Transitive group number 1 of degree 4,
 Transitive group number 2 of degree 4,
 Transitive group number 3 of degree 4,
 Transitive group number 4 of degree 4,
 Transitive group number 5 of degree 4]

sage: TransitiveGroups(5).an_element()
Transitive group number 1 of degree 5
```

```
>>> from sage.all import *
>>> S = TransitiveGroups(Integer(4)); S
Transitive Groups of degree 4
>>> list(S)
[Transitive group number 1 of degree 4,
 Transitive group number 2 of degree 4,
 Transitive group number 3 of degree 4,
 Transitive group number 4 of degree 4,
 Transitive group number 5 of degree 4]

>>> TransitiveGroups(Integer(5)).an_element()
Transitive group number 1 of degree 5
```

We write the cardinality of all transitive groups of degree 5:

```
sage: for G in TransitiveGroups(5):
....:     print(G.cardinality())
5
10
20
60
120
```

```
>>> from sage.all import *
>>> for G in TransitiveGroups(Integer(5)):
...     print(G.cardinality())
5
10
20
60
120
```

cardinality()

Return the cardinality of `self`, that is the number of transitive groups of a given degree.

EXAMPLES:

```
sage: TransitiveGroups(0).cardinality()
1
sage: TransitiveGroups(2).cardinality()
1
```

(continues on next page)

(continued from previous page)

```
sage: TransitiveGroups(7).cardinality()
7
sage: TransitiveGroups(12).cardinality()
301
sage: [TransitiveGroups(i).cardinality() for i in range(11)]
[1, 1, 1, 2, 5, 5, 16, 7, 50, 34, 45]
```

```
>>> from sage.all import *
>>> TransitiveGroups(Integer(0)).cardinality()
1
>>> TransitiveGroups(Integer(2)).cardinality()
1
>>> TransitiveGroups(Integer(7)).cardinality()
7
>>> TransitiveGroups(Integer(12)).cardinality()
301
>>> [TransitiveGroups(i).cardinality() for i in range(Integer(11))]
[1, 1, 1, 2, 5, 5, 16, 7, 50, 34, 45]
```

Warning

GAP comes with a database containing all transitive groups up to degree 31:

```
sage: TransitiveGroups(32).cardinality()
Traceback (most recent call last):
...
NotImplementedError: only the transitive groups of degree at most 31
are available in GAP's database
```

```
>>> from sage.all import *
>>> TransitiveGroups(Integer(32)).cardinality()
Traceback (most recent call last):
...
NotImplementedError: only the transitive groups of degree at most 31
are available in GAP's database
```

27.5 Permutation group elements

AUTHORS:

- David Joyner (2006-02)
- David Joyner (2006-03): word problem method and reorganization
- Robert Bradshaw (2007-11): convert to Cython
- Sebastian Oehms (2018-11): Added `gap()` as synonym to `_gap_()` (compatibility to libgap framework, see Issue #26750)
- Sebastian Oehms (2019-02): Implemented `gap()` properly (Issue #27234)

There are several ways to define a permutation group element:

- Define a permutation group G, then use `G.gens()` and multiplication `*` to construct elements.

- Define a permutation group G, then use, e.g., `G([(1,2),(3,4,5)])` to construct an element of the group. You could also use `G('(1,2)(3,4,5)')`
- Use, e.g., `PermutationGroupElement([(1,2),(3,4,5)])` or `PermutationGroupElement('(1,2)(3,4,5)')` to make a permutation group element with parent S_5.

EXAMPLES:

We illustrate construction of permutation using several different methods.

First we construct elements by multiplying together generators for a group:

```
sage: G = PermutationGroup(['(1,2)(3,4)', '(3,4,5,6)'], canonicalize=False)
sage: s = G.gens()
sage: s[0]
(1,2)(3,4)
sage: s[1]
(3,4,5,6)
sage: s[0]*s[1]
(1,2)(3,5,6)
sage: (s[0]*s[1]).parent()
Permutation Group with generators [(1,2)(3,4), (3,4,5,6)]
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2)(3,4)', '(3,4,5,6)'], canonicalize=False)
>>> s = G.gens()
>>> s[Integer(0)]
(1,2)(3,4)
>>> s[Integer(1)]
(3,4,5,6)
>>> s[Integer(0)]*s[Integer(1)]
(1,2)(3,5,6)
>>> (s[Integer(0)]*s[Integer(1)]).parent()
Permutation Group with generators [(1,2)(3,4), (3,4,5,6)]
```

Next we illustrate creation of a permutation using coercion into an already-created group:

```
sage: g = G([(1,2),(3,5,6)])
sage: g
(1,2)(3,5,6)
sage: g.parent()
Permutation Group with generators [(1,2)(3,4), (3,4,5,6)]
sage: g == s[0]*s[1]
True
```

```
>>> from sage.all import *
>>> g = G([(Integer(1),Integer(2)),(Integer(3),Integer(5),Integer(6))])
>>> g
(1,2)(3,5,6)
>>> g.parent()
Permutation Group with generators [(1,2)(3,4), (3,4,5,6)]
>>> g == s[Integer(0)]*s[Integer(1)]
True
```

We can also use a string or one-line notation to specify the permutation:

```
sage: h = G('(1,2)(3,5,6)')
sage: i = G([2,1,5,4,6,3])
sage: g == h == i
True
```

```
>>> from sage.all import *
>>> h = G('(1,2)(3,5,6)')
>>> i = G([Integer(2),Integer(1),Integer(5),Integer(4),Integer(6),Integer(3)])
>>> g == h == i
True
```

The Rubik's cube group:

```
sage: f = [(17,19,24,22),(18,21,23,20),( 6,25,43,16),( 7,28,42,13),( 8,30,41,11)]
sage: b = [(33,35,40,38),(34,37,39,36),( 3, 9,46,32),( 2,12,47,29),( 1,14,48,27)]
sage: l = [( 9,11,16,14),(10,13,15,12),( 1,17,41,40),( 4,20,44,37),( 6,22,46,35)]
sage: r = [(25,27,32,30),(26,29,31,28),( 3,38,43,19),( 5,36,45,21),( 8,33,48,24)]
sage: u = [( 1, 3, 8, 6),( 2, 5, 7, 4),( 9,33,25,17),(10,34,26,18),(11,35,27,19)]
sage: d = [(41,43,48,46),(42,45,47,44),(14,22,30,38),(15,23,31,39),(16,24,32,40)]
sage: cube = PermutationGroup([f, b, l, r, u, d])
sage: F, B, L, R, U, D = cube.gens()
sage: cube.order()
43252003274489856000
sage: F.order()
4
```

```
>>> from sage.all import *
>>> f = [(Integer(17),Integer(19),Integer(24),Integer(22)),(Integer(18),Integer(21),
↪Integer(23),Integer(20)),( Integer(6),Integer(25),Integer(43),Integer(16)),(␣
↪Integer(7),Integer(28),Integer(42),Integer(13)),( Integer(8),Integer(30),
↪Integer(41),Integer(11))]
>>> b = [(Integer(33),Integer(35),Integer(40),Integer(38)),(Integer(34),Integer(37),
↪Integer(39),Integer(36)),( Integer(3), Integer(9),Integer(46),Integer(32)),(␣
↪Integer(2),Integer(12),Integer(47),Integer(29)),( Integer(1),Integer(14),
↪Integer(48),Integer(27))]
>>> l = [( Integer(9),Integer(11),Integer(16),Integer(14)),(Integer(10),Integer(13),
↪Integer(15),Integer(12)),( Integer(1),Integer(17),Integer(41),Integer(40)),(␣
↪Integer(4),Integer(20),Integer(44),Integer(37)),( Integer(6),Integer(22),
↪Integer(46),Integer(35))]
>>> r = [(Integer(25),Integer(27),Integer(32),Integer(30)),(Integer(26),Integer(29),
↪Integer(31),Integer(28)),( Integer(3),Integer(38),Integer(43),Integer(19)),(␣
↪Integer(5),Integer(36),Integer(45),Integer(21)),( Integer(8),Integer(33),
↪Integer(48),Integer(24))]
>>> u = [( Integer(1), Integer(3), Integer(8), Integer(6)),( Integer(2), Integer(5),␣
↪Integer(7), Integer(4)),( Integer(9),Integer(33),Integer(25),Integer(17)),
↪(Integer(10),Integer(34),Integer(26),Integer(18)),(Integer(11),Integer(35),
↪Integer(27),Integer(19))]
>>> d = [(Integer(41),Integer(43),Integer(48),Integer(46)),(Integer(42),Integer(45),
↪Integer(47),Integer(44)),(Integer(14),Integer(22),Integer(30),Integer(38)),
↪(Integer(15),Integer(23),Integer(31),Integer(39)),(Integer(16),Integer(24),
↪Integer(32),Integer(40))]
>>> cube = PermutationGroup([f, b, l, r, u, d])
```

(continues on next page)

(continued from previous page)

```
>>> F, B, L, R, U, D = cube.gens()
>>> cube.order()
43252003274489856000
>>> F.order()
4
```

We create element of a permutation group of large degree:

```
sage: G = SymmetricGroup(30)
sage: s = G(srange(30,0,-1)); s
(1,30)(2,29)(3,28)(4,27)(5,26)(6,25)(7,24)(8,23)(9,22)(10,21)(11,20)(12,19)(13,18)(14,
↪17)(15,16)
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(30))
>>> s = G(srange(Integer(30),Integer(0),-Integer(1))); s
(1,30)(2,29)(3,28)(4,27)(5,26)(6,25)(7,24)(8,23)(9,22)(10,21)(11,20)(12,19)(13,18)(14,
↪17)(15,16)
```

class sage.groups.perm_gps.permgroup_element.**PermutationGroupElement**

Bases: MultiplicativeGroupElement

An element of a permutation group.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)']); G
Permutation Group with generators [(1,2,3)(4,5)]
sage: g = G.random_element()
sage: g in G
True
sage: g = G.gen(0); g
(1,2,3)(4,5)
sage: print(g)
(1,2,3)(4,5)
sage: g*g
(1,3,2)
sage: g**(-1)
(1,3,2)(4,5)
sage: g**2
(1,3,2)
sage: G = PermutationGroup([(1,2,3)])
sage: g = G.gen(0); g
(1,2,3)
sage: g.order()
3
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)']); G
Permutation Group with generators [(1,2,3)(4,5)]
>>> g = G.random_element()
>>> g in G
True
```

(continues on next page)

(continued from previous page)

```
>>> g = G.gen(Integer(0)); g
(1,2,3)(4,5)
>>> print(g)
(1,2,3)(4,5)
>>> g*g
(1,3,2)
>>> g**(-Integer(1))
(1,3,2)(4,5)
>>> g**Integer(2)
(1,3,2)
>>> G = PermutationGroup([(Integer(1),Integer(2),Integer(3))])
>>> g = G.gen(Integer(0)); g
(1,2,3)
>>> g.order()
3
```

This example illustrates how permutations act on multivariate polynomials.

```
sage: R = PolynomialRing(RationalField(), 5, ["x","y","z","u","v"])
sage: x, y, z, u, v = R.gens()
sage: f = x**2 - y**2 + 3*z**2
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: sigma = G.gen(0)
sage: f * sigma
3*x^2 + y^2 - z^2
```

```
>>> from sage.all import *
>>> R = PolynomialRing(RationalField(), Integer(5), ["x","y","z","u","v"])
>>> x, y, z, u, v = R.gens()
>>> f = x**Integer(2) - y**Integer(2) + Integer(3)*z**Integer(2)
>>> G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
>>> sigma = G.gen(Integer(0))
>>> f * sigma
3*x^2 + y^2 - z^2
```

cycle_string(*singletons=False*)

Return string representation of this permutation.

EXAMPLES:

```
sage: g = PermutationGroupElement([(1,2,3),(4,5)])
sage: g.cycle_string()
'(1,2,3)(4,5)'

sage: g = PermutationGroupElement([3,2,1])
sage: g.cycle_string(singletons=True)
'(1,3)(2)'
```

```
>>> from sage.all import *
>>> g = PermutationGroupElement([(Integer(1),Integer(2),Integer(3)),
↪(Integer(4),Integer(5))])
>>> g.cycle_string()
```

(continues on next page)

(continued from previous page)

```
'(1,2,3)(4,5)'

>>> g = PermutationGroupElement([Integer(3),Integer(2),Integer(1)])
>>> g.cycle_string(singletons=True)
'(1,3)(2)'
```

cycle_tuples(*singletons=False*)

Return `self` as a list of disjoint cycles, represented as tuples rather than permutation group elements.

INPUT:

- `singletons` – boolean (default: `False`); whether or not consider the cycle that correspond to fixed point

EXAMPLES:

```
sage: p = PermutationGroupElement('(2,6)(4,5,1)')
sage: p.cycle_tuples()
[(1, 4, 5), (2, 6)]
sage: p.cycle_tuples(singletons=True)
[(1, 4, 5), (2, 6), (3,)]
```

```
>>> from sage.all import *
>>> p = PermutationGroupElement('(2,6)(4,5,1)')
>>> p.cycle_tuples()
[(1, 4, 5), (2, 6)]
>>> p.cycle_tuples(singletons=True)
[(1, 4, 5), (2, 6), (3,)]
```

EXAMPLES:

```
sage: S = SymmetricGroup(4)
sage: S.gen(0).cycle_tuples()
[(1, 2, 3, 4)]
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(Integer(4))
>>> S.gen(Integer(0)).cycle_tuples()
[(1, 2, 3, 4)]
```

```
sage: S = SymmetricGroup(['a','b','c','d'])
sage: S.gen(0).cycle_tuples()
[('a', 'b', 'c', 'd')]
sage: S([('a', 'b'), ('c', 'd')]).cycle_tuples()
[('a', 'b'), ('c', 'd')]
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(['a','b','c','d'])
>>> S.gen(Integer(0)).cycle_tuples()
[('a', 'b', 'c', 'd')]
>>> S([('a', 'b'), ('c', 'd')]).cycle_tuples()
[('a', 'b'), ('c', 'd')]
```

cycle_type(*singletons=True*, *as_list=False*)

Return the partition that gives the cycle type of `self` as an element of its parent.

INPUT:

- `g` – an element of the permutation group `self.parent()`
- `singletons` – boolean depending on whether or not trivial cycles should be counted (default: `True`)
- `as_list` – boolean depending on whether the cycle type should be returned as a `list` or as a `Partition` (default: `False`)

OUTPUT: a `Partition`, or `list` if `is_list` is `True`, giving the cycle type of `self`

If speed is a concern, then `as_list=True` should be used.

EXAMPLES:

```
sage: # needs sage.combinat
sage: G = DihedralGroup(3)
sage: [g.cycle_type() for g in G]
[[1, 1, 1], [3], [3], [2, 1], [2, 1], [2, 1]]
sage: PermutationGroupElement('(1,2,3)(4,5)(6,7,8)').cycle_type()
[3, 3, 2]
sage: G = SymmetricGroup(3); G('(1,2)').cycle_type()
[2, 1]
sage: G = SymmetricGroup(4); G('(1,2)').cycle_type()
[2, 1, 1]
sage: G = SymmetricGroup(4); G('(1,2)').cycle_type(singletons=False)
[2]
sage: G = SymmetricGroup(4); G('(1,2)').cycle_type(as_list=False)
[2, 1, 1]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> G = DihedralGroup(Integer(3))
>>> [g.cycle_type() for g in G]
[[1, 1, 1], [3], [3], [2, 1], [2, 1], [2, 1]]
>>> PermutationGroupElement('(1,2,3)(4,5)(6,7,8)').cycle_type()
[3, 3, 2]
>>> G = SymmetricGroup(Integer(3)); G('(1,2)').cycle_type()
[2, 1]
>>> G = SymmetricGroup(Integer(4)); G('(1,2)').cycle_type()
[2, 1, 1]
>>> G = SymmetricGroup(Integer(4)); G('(1,2)').cycle_type(singletons=False)
[2]
>>> G = SymmetricGroup(Integer(4)); G('(1,2)').cycle_type(as_list=False)
[2, 1, 1]
```

cycles()

Return `self` as a list of disjoint cycles.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5,6,7)'])
sage: g = G.0
sage: g.cycles()
```

(continues on next page)

(continued from previous page)

```
[(1,2,3), (4,5,6,7)]
sage: a, b = g.cycles()
sage: a(1), b(1)
(2, 1)
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5,6,7)'])
>>> g = G.gen(0)
>>> g.cycles()
[(1,2,3), (4,5,6,7)]
>>> a, b = g.cycles()
>>> a(Integer(1)), b(Integer(1))
(2, 1)
```

dict()

Return a dictionary associating each element of the domain with its image.

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4)); g
(1,2,3,4)
sage: v = g.dict(); v
{1: 2, 2: 3, 3: 4, 4: 1}
sage: type(v[1])
<... 'int'>
sage: x = G([2,1]); x
(1,2)
sage: x.dict()
{1: 2, 2: 1, 3: 3, 4: 4}
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
>>> g = G((Integer(1),Integer(2),Integer(3),Integer(4))); g
(1,2,3,4)
>>> v = g.dict(); v
{1: 2, 2: 3, 3: 4, 4: 1}
>>> type(v[Integer(1)])
<... 'int'>
>>> x = G([Integer(2),Integer(1)]); x
(1,2)
>>> x.dict()
{1: 2, 2: 1, 3: 3, 4: 4}
```

domain()

Return the domain of `self`.

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: x = G([2,1,4,3]); x
(1,2)(3,4)
```

(continues on next page)

(continued from previous page)

```
sage: v = x.domain(); v
[2, 1, 4, 3]
sage: type(v[0])
<... 'int'>
sage: x = G([2,1]); x
(1,2)
sage: x.domain()
[2, 1, 3, 4]
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
>>> x = G([Integer(2),Integer(1),Integer(4),Integer(3)]); x
(1,2)(3,4)
>>> v = x.domain(); v
[2, 1, 4, 3]
>>> type(v[Integer(0)])
<... 'int'>
>>> x = G([Integer(2),Integer(1)]); x
(1,2)
>>> x.domain()
[2, 1, 3, 4]
```

gap()

Return `self` as a libgap element.

EXAMPLES:

```
sage: S = SymmetricGroup(4)
sage: p = S('(2,4)')
sage: p_libgap = libgap(p)
sage: p_libgap.Order()
2
sage: S(p_libgap) == p
True

sage: # needs sage.rings.finite_rings
sage: P = PGU(8,2)
sage: p, q = P.gens()
sage: p_libgap  = p.gap()
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(Integer(4))
>>> p = S('(2,4)')
>>> p_libgap = libgap(p)
>>> p_libgap.Order()
2
>>> S(p_libgap) == p
True

>>> # needs sage.rings.finite_rings
>>> P = PGU(Integer(8),Integer(2))
>>> p, q = P.gens()
```

(continues on next page)

(continued from previous page)

```
>>> p_libgap  = p.gap()
```

has_descent(*i*, *side='right'*, *positive=False*)

Return whether `self` has a left (resp. right) descent at position `i`. If `positive` is `True`, then test for a non descent instead.

Beware that, since permutations are acting on the right, the meaning of descents is the reverse of the usual convention. Hence, `self` has a left descent at position `i` if `self(i) > self(i+1)`.

INPUT:

- `i` – an element of the index set
- `side` – `'left'` or `'right'` (default: `'right'`)
- `positive` – boolean (default: `False`)

EXAMPLES:

```
sage: S = SymmetricGroup([1,2,3])
sage: S.one().has_descent(1)
False
sage: S.one().has_descent(2)
False
sage: s = S.simple_reflections()
sage: x = s[1]*s[2]
sage: x.has_descent(1, side='right')
False
sage: x.has_descent(2, side='right')
True
sage: x.has_descent(1, side='left')
True
sage: x.has_descent(2, side='left')
False
sage: S._test_has_descent()
```

```
>>> from sage.all import *
>>> S = SymmetricGroup([Integer(1),Integer(2),Integer(3)])
>>> S.one().has_descent(Integer(1))
False
>>> S.one().has_descent(Integer(2))
False
>>> s = S.simple_reflections()
>>> x = s[Integer(1)]*s[Integer(2)]
>>> x.has_descent(Integer(1), side='right')
False
>>> x.has_descent(Integer(2), side='right')
True
>>> x.has_descent(Integer(1), side='left')
True
>>> x.has_descent(Integer(2), side='left')
False
>>> S._test_has_descent()
```

The symmetric group acting on a set not of the form $(1, \dots, n)$ is also supported:

```
sage: S = SymmetricGroup([2,4,1])
sage: s = S.simple_reflections()
sage: x = s[2]*s[4]
sage: x.has_descent(4)
True
sage: S._test_has_descent()
```

```
>>> from sage.all import *
>>> S = SymmetricGroup([Integer(2),Integer(4),Integer(1)])
>>> s = S.simple_reflections()
>>> x = s[Integer(2)]*s[Integer(4)]
>>> x.has_descent(Integer(4))
True
>>> S._test_has_descent()
```

inverse()

Return the inverse permutation.

OUTPUT:

For an element of a permutation group, this method returns the inverse element, which is both the inverse function and the inverse as an element of a group.

EXAMPLES:

```
sage: s = PermutationGroupElement("(1,2,3)(4,5)")
sage: s.inverse()
(1,3,2)(4,5)

sage: A = AlternatingGroup(4)
sage: t = A("(1,2,3)")
sage: t.inverse()
(1,3,2)
```

```
>>> from sage.all import *
>>> s = PermutationGroupElement("(1,2,3)(4,5)")
>>> s.inverse()
(1,3,2)(4,5)

>>> A = AlternatingGroup(Integer(4))
>>> t = A("(1,2,3)")
>>> t.inverse()
(1,3,2)
```

There are several ways (syntactically) to get an inverse of a permutation group element.

```
sage: s = PermutationGroupElement("(1,2,3,4)(6,7,8)")
sage: s.inverse() == s^-1
True
sage: s.inverse() == ~s
True
```

```
>>> from sage.all import *
>>> s = PermutationGroupElement("(1,2,3,4)(6,7,8)")
```

(continues on next page)

(continued from previous page)

```
>>> s.inverse() == s**-Integer(1)
True
>>> s.inverse() == ~s
True
```

matrix()

Return a deg $\times$ deg permutation matrix associated to the permutation `self`.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: g = G.gen(0)
sage: g.matrix()
[0 1 0 0 0]
[0 0 1 0 0]
[1 0 0 0 0]
[0 0 0 0 1]
[0 0 0 1 0]
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)'])
>>> g = G.gen(Integer(0))
>>> g.matrix()
[0 1 0 0 0]
[0 0 1 0 0]
[1 0 0 0 0]
[0 0 0 0 1]
[0 0 0 1 0]
```

multiplicative_order()

Return the order of this group element, which is the smallest positive integer n for which $g^n = 1$.

EXAMPLES:

```
sage: s = PermutationGroupElement('(1,2)(3,5,6)')
sage: s.multiplicative_order()
6
```

```
>>> from sage.all import *
>>> s = PermutationGroupElement('(1,2)(3,5,6)')
>>> s.multiplicative_order()
6
```

`order()` is just an alias for *`multiplicative_order()`*:

```
sage: s.order()
6
```

```
>>> from sage.all import *
>>> s.order()
6
```

orbit(*n*, *sorted=True*)

Return the orbit of the integer n under this group element, as a sorted list.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: g = G.gen(0)
sage: g.orbit(4)
[4, 5]
sage: g.orbit(3)
[1, 2, 3]
sage: g.orbit(10)
[10]
```

```
>>> from sage.all import *
>>> G = PermutationGroup(['(1,2,3)(4,5)'])
>>> g = G.gen(Integer(0))
>>> g.orbit(Integer(4))
[4, 5]
>>> g.orbit(Integer(3))
[1, 2, 3]
>>> g.orbit(Integer(10))
[10]
```

```
sage: s = SymmetricGroup(['a', 'b']).gen(0); s
('a','b')
sage: s.orbit('a')
['a', 'b']
```

```
>>> from sage.all import *
>>> s = SymmetricGroup(['a', 'b']).gen(Integer(0)); s
('a','b')
>>> s.orbit('a')
['a', 'b']
```

sign()

Return the sign of `self`, which is $(-1)^s$, where s is the number of swaps.

EXAMPLES:

```
sage: s = PermutationGroupElement('(1,2)(3,5,6)')
sage: s.sign()
-1
```

```
>>> from sage.all import *
>>> s = PermutationGroupElement('(1,2)(3,5,6)')
>>> s.sign()
-1
```

ALGORITHM: Only even cycles contribute to the sign, thus

$$sign(sigma) = (-1)^{\sum_c len(c)-1}$$

where the sum is over cycles in `self`.

tuple()

Return tuple of images of the domain under `self`.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: s = G([2,1,5,3,4])
sage: s.tuple()
(2, 1, 5, 3, 4)

sage: S = SymmetricGroup(['a', 'b'])
sage: S.gen().tuple()
('b', 'a')
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> s = G([Integer(2),Integer(1),Integer(5),Integer(3),Integer(4)])
>>> s.tuple()
(2, 1, 5, 3, 4)

>>> S = SymmetricGroup(['a', 'b'])
>>> S.gen().tuple()
('b', 'a')
```

word_problem(*words*, *display=True*, *as_list=False*)

Try to solve the word problem for `self`.

INPUT:

- `words` – list of elements of the ambient group, generating a subgroup
- `display` – boolean (default: `True`); whether to display additional information
- `as_list` – boolean (default: `False`); whether to return the result as a list of pairs (generator, exponent)

OUTPUT:

- a pair of strings, both representing the same word

or

- a list of pairs representing the word, each pair being (generator as a string, exponent as an integer)

Let G be the ambient permutation group, containing the given element g. Let H be the subgroup of G generated by the list `words` of elements of G. If g is in H, this function returns an expression for g as a word in the elements of `words` and their inverses.

This function does not solve the word problem in Sage. Rather it pushes it over to GAP, which has optimized algorithms for the word problem. Essentially, this function is a wrapper for the GAP functions `EpimorphismFromFreeGroup` and `PreImagesRepresentative`.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]], canonicalize=False)
sage: g1, g2 = G.gens()
sage: h = g1^2*g2*g1
sage: h.word_problem([g1,g2], False)
('x1^2*x2^-1*x1', '(1,2,3)(4,5)^2*(3,4)^-1*(1,2,3)(4,5)')
```

(continues on next page)

(continued from previous page)

```
sage: h.word_problem([g1,g2])
   x1^2*x2^-1*x1
   [['(1,2,3)(4,5)', 2], ['(3,4)', -1], ['(1,2,3)(4,5)', 1]]
('x1^2*x2^-1*x1', '(1,2,3)(4,5)^2*(3,4)^-1*(1,2,3)(4,5)')

sage: h.word_problem([g1,g2], False, as_list=True)
[['(1,2,3)(4,5)', 2], ['(3,4)', -1], ['(1,2,3)(4,5)', 1]]
```

```
>>> from sage.all import *
>>> G = PermutationGroup([[(Integer(1),Integer(2),Integer(3)),(Integer(4),
↪Integer(5))],[(Integer(3),Integer(4))]], canonicalize=False)
>>> g1, g2 = G.gens()
>>> h = g1**Integer(2)*g2*g1
>>> h.word_problem([g1,g2], False)
('x1^2*x2^-1*x1', '(1,2,3)(4,5)^2*(3,4)^-1*(1,2,3)(4,5)')

>>> h.word_problem([g1,g2])
   x1^2*x2^-1*x1
   [['(1,2,3)(4,5)', 2], ['(3,4)', -1], ['(1,2,3)(4,5)', 1]]
('x1^2*x2^-1*x1', '(1,2,3)(4,5)^2*(3,4)^-1*(1,2,3)(4,5)')

>>> h.word_problem([g1,g2], False, as_list=True)
[['(1,2,3)(4,5)', 2], ['(3,4)', -1], ['(1,2,3)(4,5)', 1]]
```

class sage.groups.perm_gps.permgroup_element.**SymmetricGroupElement**

Bases: *PermutationGroupElement*

An element of the symmetric group.

absolute_length()

Return the absolute length of `self`.

The absolute length is the size minus the number of its disjoint cycles. Alternatively, it is the length of the shortest expression of the element as a product of reflections.

See also

`absolute_le()`

EXAMPLES:

```
sage: S = SymmetricGroup(3)
sage: [x.absolute_length() for x in S]                                    #␣
↪needs sage.combinat
[0, 2, 2, 1, 1, 1]
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(Integer(3))
>>> [x.absolute_length() for x in S]                                    #␣
↪needs sage.combinat
[0, 2, 2, 1, 1, 1]
```

has_left_descent(*i*)

Return whether i is a left descent of `self`.

EXAMPLES:

```
sage: W = SymmetricGroup(4)
sage: w = W.from_reduced_word([1,3,2,1])
sage: [i for i in W.index_set() if w.has_left_descent(i)]
[1, 3]
```

```
>>> from sage.all import *
>>> W = SymmetricGroup(Integer(4))
>>> w = W.from_reduced_word([Integer(1),Integer(3),Integer(2),Integer(1)])
>>> [i for i in W.index_set() if w.has_left_descent(i)]
[1, 3]
```

`sage.groups.perm_gps.permgroup_element.`**is_PermutationGroupElement**(*x*)

Return `True` if `x` is a *PermutationGroupElement*.

EXAMPLES:

```
sage: p = PermutationGroupElement([(1,2),(3,4,5)])
sage: from sage.groups.perm_gps.permgroup_element import is_
↪PermutationGroupElement
sage: is_PermutationGroupElement(p)
doctest:warning...
DeprecationWarning: The function is_PermutationGroupElement is deprecated;
use 'isinstance(..., PermutationGroupElement)' instead.
See https://github.com/sagemath/sage/issues/38184 for details.
True
```

```
>>> from sage.all import *
>>> p = PermutationGroupElement([(Integer(1),Integer(2)),(Integer(3),Integer(4),
↪Integer(5))])
>>> from sage.groups.perm_gps.permgroup_element import is_PermutationGroupElement
>>> is_PermutationGroupElement(p)
doctest:warning...
DeprecationWarning: The function is_PermutationGroupElement is deprecated;
use 'isinstance(..., PermutationGroupElement)' instead.
See https://github.com/sagemath/sage/issues/38184 for details.
True
```

`sage.groups.perm_gps.permgroup_element.`**make_permgroup_element**(*G*, *x*)

Return a *PermutationGroupElement* given the permutation group `G` and the permutation `x` in list notation.

This is function is used when unpickling old (pre-domain) versions of permutation groups and their elements. This now does a bit of processing and calls *make_permgroup_element_v2()* which is used in unpickling the current PermutationGroupElements.

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup_element import make_permgroup_element
sage: S = SymmetricGroup(3)
sage: make_permgroup_element(S, [1,3,2])
(2,3)
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.permgroup_element import make_permgroup_element
>>> S = SymmetricGroup(Integer(3))
>>> make_permgroup_element(S, [Integer(1),Integer(3),Integer(2)])
(2,3)
```

sage.groups.perm_gps.permgroup_element.**make_permgroup_element_v2**(*G*, *x*, *domain*)

Return a `PermutationGroupElement` given the permutation group `G`, the permutation `x` in list notation, and the domain `domain` of the permutation group.

This is function is used when unpickling permutation groups and their elements.

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup_element import make_permgroup_element_v2
sage: S = SymmetricGroup(3)
sage: make_permgroup_element_v2(S, [1,3,2], S.domain())
(2,3)
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.permgroup_element import make_permgroup_element_v2
>>> S = SymmetricGroup(Integer(3))
>>> make_permgroup_element_v2(S, [Integer(1),Integer(3),Integer(2)], S.domain())
(2,3)
```

27.6 Permutation group homomorphisms

AUTHORS:

- David Joyner (2006-03-21): first version
- David Joyner (2008-06): fixed kernel and image to return a group, instead of a string.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: H = DihedralGroup(4)
sage: g = G([(1,2,3,4)])
sage: phi = PermutationGroupMorphism_im_gens(G, H, map(H, G.gens()))
sage: phi.image(G)
Subgroup generated by [(1,2,3,4)] of
 (Dihedral group of order 8 as a permutation group)
sage: phi.kernel()
Subgroup generated by [()] of (Cyclic group of order 4 as a permutation group)
sage: phi.image(g)
(1,2,3,4)
sage: phi(g)
(1,2,3,4)
sage: phi.codomain()
Dihedral group of order 8 as a permutation group
sage: phi.codomain()
Dihedral group of order 8 as a permutation group
sage: phi.domain()
Cyclic group of order 4 as a permutation group
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(4))
>>> H = DihedralGroup(Integer(4))
>>> g = G([(Integer(1),Integer(2),Integer(3),Integer(4))])
>>> phi = PermutationGroupMorphism_im_gens(G, H, map(H, G.gens()))
>>> phi.image(G)
Subgroup generated by [(1,2,3,4)] of
 (Dihedral group of order 8 as a permutation group)
>>> phi.kernel()
Subgroup generated by [()] of (Cyclic group of order 4 as a permutation group)
>>> phi.image(g)
(1,2,3,4)
>>> phi(g)
(1,2,3,4)
>>> phi.codomain()
Dihedral group of order 8 as a permutation group
>>> phi.codomain()
Dihedral group of order 8 as a permutation group
>>> phi.domain()
Cyclic group of order 4 as a permutation group
```

class sage.groups.perm_gps.permgroup_morphism.**PermutationGroupMorphism**

Bases: `Morphism`

A set-theoretic map between PermutationGroups.

image(J)

Compute the subgroup of the codomain H which is the image of J.

J must be a subgroup of the domain G.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: H = DihedralGroup(4)
sage: g = G([(1,2,3,4)])
sage: phi = PermutationGroupMorphism_im_gens(G, H, map(H, G.gens()))
sage: phi.image(G)
Subgroup generated by [(1,2,3,4)] of
 (Dihedral group of order 8 as a permutation group)
sage: phi.image(g)
(1,2,3,4)
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(4))
>>> H = DihedralGroup(Integer(4))
>>> g = G([(Integer(1),Integer(2),Integer(3),Integer(4))])
>>> phi = PermutationGroupMorphism_im_gens(G, H, map(H, G.gens()))
>>> phi.image(G)
Subgroup generated by [(1,2,3,4)] of
 (Dihedral group of order 8 as a permutation group)
>>> phi.image(g)
(1,2,3,4)
```

```
sage: G = PSL(2,7)
sage: D = G.direct_product(G)
sage: H = D[0]
sage: pr1 = D[3]
sage: pr1.image(G)
Subgroup generated by [(3,7,5)(4,8,6), (1,2,6)(3,4,8)] of
 (The projective special linear group of degree 2 over Finite Field of size 7)
sage: G.is_isomorphic(pr1.image(G))
True
```

```
>>> from sage.all import *
>>> G = PSL(Integer(2),Integer(7))
>>> D = G.direct_product(G)
>>> H = D[Integer(0)]
>>> pr1 = D[Integer(3)]
>>> pr1.image(G)
Subgroup generated by [(3,7,5)(4,8,6), (1,2,6)(3,4,8)] of
 (The projective special linear group of degree 2 over Finite Field of size 7)
>>> G.is_isomorphic(pr1.image(G))
True
```

Check that Issue #28324 is fixed:

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: f = x^4 + x^2 - 3
sage: L.<a> = f.splitting_field()
sage: G = L.galois_group()
sage: D4 = DihedralGroup(4)
sage: h = D4.isomorphism_to(G)
sage: h.image(D4).is_isomorphic(G)
True
sage: all(h.image(g) in G for g in D4.gens())
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> f = x**Integer(4) + x**Integer(2) - Integer(3)
>>> L = f.splitting_field(names=('a',)); (a,) = L._first_ngens(1)
>>> G = L.galois_group()
>>> D4 = DihedralGroup(Integer(4))
>>> h = D4.isomorphism_to(G)
>>> h.image(D4).is_isomorphic(G)
True
>>> all(h.image(g) in G for g in D4.gens())
True
```

kernel()

Return the kernel of this homomorphism as a permutation group.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: H = DihedralGroup(4)
sage: g = G([(1,2,3,4)])
sage: phi = PermutationGroupMorphism_im_gens(G, H, [1])
sage: phi.kernel()
Subgroup generated by [(1,2,3,4)] of
 (Cyclic group of order 4 as a permutation group)
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(4))
>>> H = DihedralGroup(Integer(4))
>>> g = G([(Integer(1),Integer(2),Integer(3),Integer(4))])
>>> phi = PermutationGroupMorphism_im_gens(G, H, [Integer(1)])
>>> phi.kernel()
Subgroup generated by [(1,2,3,4)] of
 (Cyclic group of order 4 as a permutation group)
```

```
sage: G = PSL(2,7)
sage: D = G.direct_product(G)
sage: H = D[0]
sage: pr1 = D[3]
sage: G.is_isomorphic(pr1.kernel())
True
```

```
>>> from sage.all import *
>>> G = PSL(Integer(2),Integer(7))
>>> D = G.direct_product(G)
>>> H = D[Integer(0)]
>>> pr1 = D[Integer(3)]
>>> G.is_isomorphic(pr1.kernel())
True
```

class `sage.groups.perm_gps.permgroup_morphism.`**`PermutationGroupMorphism_from_gap`**(G, H, gap_hom)

Bases: `PermutationGroupMorphism`

This is a Python trick to allow Sage programmers to create a group homomorphism using GAP using very general constructions. An example of its usage is in the direct_product instance method of the `PermutationGroup_generic` class in permgroup.py.

Basic syntax:

`PermutationGroupMorphism_from_gap(domain_group, range_group, 'phi:=gap_hom_command;', 'phi')`. And don't forget the line: `from sage.groups.perm_gps.permgroup_morphism import PermutationGroupMorphism_from_gap` in your program.

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup_morphism import
↪PermutationGroupMorphism_from_gap
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: H = G.subgroup([G([(1,2,3,4)])])
sage: PermutationGroupMorphism_from_gap(H, G, gap.Identity)
Permutation group morphism:
```

(continues on next page)

(continued from previous page)

```
  From: Subgroup generated by [(1,2,3,4)] of
        (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
  To:   Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]
  Defn: Identity
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.permgroup_morphism import PermutationGroupMorphism_
↪from_gap
>>> G = PermutationGroup([[(Integer(1),Integer(2)),(Integer(3),Integer(4))],␣
↪[(Integer(1),Integer(2),Integer(3),Integer(4))]])
>>> H = G.subgroup([G([(Integer(1),Integer(2),Integer(3),Integer(4))])])
>>> PermutationGroupMorphism_from_gap(H, G, gap.Identity)
Permutation group morphism:
  From: Subgroup generated by [(1,2,3,4)] of
        (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
  To:   Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]
  Defn: Identity
```

class sage.groups.perm_gps.permgroup_morphism.**PermutationGroupMorphism_id**

Bases: *PermutationGroupMorphism*

class sage.groups.perm_gps.permgroup_morphism.**PermutationGroupMorphism_im_gens**(*G*, *H*, *gens=None*)

Bases: *PermutationGroupMorphism*

Some python code for wrapping GAP's `GroupHomomorphismByImages` function but only for permutation groups. Can be expensive if G is large. This returns "fail" if gens does not generate `self` or if the map does not extend to a group homomorphism, `self` - `other`.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: H = DihedralGroup(4)
sage: phi = PermutationGroupMorphism_im_gens(G, H, map(H, G.gens())); phi
Permutation group morphism:
  From: Cyclic group of order 4 as a permutation group
  To:   Dihedral group of order 8 as a permutation group
  Defn: [(1,2,3,4)] -> [(1,2,3,4)]
sage: g = G([(1,3),(2,4)]); g
(1,3)(2,4)
sage: phi(g)
(1,3)(2,4)
sage: images = ((4,3,2,1),)
sage: phi = PermutationGroupMorphism_im_gens(G, G, images)
sage: g = G([(1,2,3,4)]); g
(1,2,3,4)
sage: phi(g)
(1,4,3,2)
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(4))
>>> H = DihedralGroup(Integer(4))
>>> phi = PermutationGroupMorphism_im_gens(G, H, map(H, G.gens())); phi
```

(continues on next page)

(continued from previous page)

```
Permutation group morphism:
  From: Cyclic group of order 4 as a permutation group
  To:   Dihedral group of order 8 as a permutation group
  Defn: [(1,2,3,4)] -> [(1,2,3,4)]
>>> g = G([(Integer(1),Integer(3)),(Integer(2),Integer(4))]); g
(1,3)(2,4)
>>> phi(g)
(1,3)(2,4)
>>> images = ((Integer(4),Integer(3),Integer(2),Integer(1)),)
>>> phi = PermutationGroupMorphism_im_gens(G, G, images)
>>> g = G([(Integer(1),Integer(2),Integer(3),Integer(4))]); g
(1,2,3,4)
>>> phi(g)
(1,4,3,2)
```

AUTHORS:

- David Joyner (2006-02)

`sage.groups.perm_gps.permgroup_morphism.`**`is_PermutationGroupMorphism`**`(f)`

Return `True` if the argument `f` is a *PermutationGroupMorphism*.

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup_morphism import is_
↪PermutationGroupMorphism
sage: G = CyclicPermutationGroup(4)
sage: H = DihedralGroup(4)
sage: phi = PermutationGroupMorphism_im_gens(G, H, map(H, G.gens()))
sage: is_PermutationGroupMorphism(phi)
doctest:warning...
DeprecationWarning: The function is_PermutationGroupMorphism is deprecated;
use 'isinstance(..., PermutationGroupMorphism)' instead.
See https://github.com/sagemath/sage/issues/38103 for details.
True
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.permgroup_morphism import is_
↪PermutationGroupMorphism
>>> G = CyclicPermutationGroup(Integer(4))
>>> H = DihedralGroup(Integer(4))
>>> phi = PermutationGroupMorphism_im_gens(G, H, map(H, G.gens()))
>>> is_PermutationGroupMorphism(phi)
doctest:warning...
DeprecationWarning: The function is_PermutationGroupMorphism is deprecated;
use 'isinstance(..., PermutationGroupMorphism)' instead.
See https://github.com/sagemath/sage/issues/38103 for details.
True
```

27.7 Rubik's cube group functions

Note

"Rubiks cube" is trademarked. We shall omit the trademark symbol below for simplicity.

NOTATION:

B denotes a clockwise quarter turn of the back face, D denotes a clockwise quarter turn of the down face, and similarly for F (front), L (left), R (right), and U (up). Products of moves are read right to left, so for example, $R \cdot U$ means move U first and then R.

See `CubeGroup.parse()` for all possible input notations.

The "Singmaster notation":

- moves: U, D, R, L, F, B as in the diagram below,
- corners: xyz means the facet is on face x (in R, F, L, U, D, B) and the clockwise rotation of the corner sends $x - y - z$
- edges: xy means the facet is on face x and a flip of the edge sends $x - y$.

```
sage: rubik = CubeGroup()
sage: rubik.display2d("")
             ┌──────────────┐
             │  1    2    3 │
             │  4   top   5 │
             │  6    7    8 │
┌────────────┼──────────────┼─────────────┬────────────┐
│  9  10  11 │ 17   18   19 │ 25   26  27 │ 33  34  35 │
│ 12 left 13 │ 20  front 21 │ 28 right 29 │ 36 rear 37 │
│ 14  15  16 │ 22   23   24 │ 30   31  32 │ 38  39  40 │
└────────────┼──────────────┼─────────────┴────────────┘
             │ 41   42   43 │
             │ 44 bottom 45 │
             │ 46   47   48 │
             └──────────────┘
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> rubik.display2d("")
             ┌──────────────┐
             │  1    2    3 │
             │  4   top   5 │
             │  6    7    8 │
┌────────────┼──────────────┼─────────────┬────────────┐
│  9  10  11 │ 17   18   19 │ 25   26  27 │ 33  34  35 │
│ 12 left 13 │ 20  front 21 │ 28 right 29 │ 36 rear 37 │
│ 14  15  16 │ 22   23   24 │ 30   31  32 │ 38  39  40 │
└────────────┼──────────────┼─────────────┴────────────┘
             │ 41   42   43 │
             │ 44 bottom 45 │
             │ 46   47   48 │
             └──────────────┘
```

AUTHORS:

- David Joyner (2006-10-21): first version
- David Joyner (2007-05): changed faces, added legal and solve
- David Joyner(2007-06): added plotting functions
- David Joyner (2007, 2008): colors corrected, "solve" rewritten (again),typos fixed.
- Robert Miller (2007, 2008): editing, cleaned up display2d
- Robert Bradshaw (2007, 2008): RubiksCube object, 3d plotting.
- David Joyner (2007-09): rewrote docstring for CubeGroup's "solve".
- Robert Bradshaw (2007-09): Versatile parse function for all input types.
- Robert Bradshaw (2007-11): Cleanup.

REFERENCES:

- Cameron, P., Permutation Groups. New York: Cambridge University Press, 1999.
- Wielandt, H., Finite Permutation Groups. New York: Academic Press, 1964.
- Dixon, J. and Mortimer, B., Permutation Groups, Springer-Verlag, Berlin/New York, 1996.
- Joyner, D., Adventures in Group Theory, Johns Hopkins Univ Press, 2002.

class sage.groups.perm_gps.cubegroup.**CubeGroup**

Bases: *PermutationGroup_generic*

A python class to help compute Rubik's cube group actions.

Note

This group is also available via `groups.permutation.RubiksCube()`.

EXAMPLES:

If G denotes the cube group then it may be regarded as a subgroup of `SymmetricGroup(48)`, where the 48 facets are labeled as follows.

```
sage: rubik = CubeGroup()
sage: rubik.display2d("")
             +--------------+
             |  1    2    3 |
             |  4   top   5 |
             |  6    7    8 |
+------------+--------------+-------------+------------+
|  9  10  11 | 17   18   19 | 25   26  27 | 33  34  35 |
| 12 left 13 | 20  front 21 | 28 right 29 | 36 rear 37 |
| 14  15  16 | 22   23   24 | 30   31  32 | 38  39  40 |
+------------+--------------+-------------+------------+
             | 41   42   43 |
             | 44 bottom 45 |
             | 46   47   48 |
             +--------------+
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> rubik.display2d("")
             +--------------+
             |  1    2    3 |
             |  4   top   5 |
             |  6    7    8 |
+------------+--------------+-------------+------------+
|  9  10  11 | 17   18   19 | 25   26  27 | 33  34  35 |
| 12 left 13 | 20  front 21 | 28 right 29 | 36 rear 37 |
| 14  15  16 | 22   23   24 | 30   31  32 | 38  39  40 |
+------------+--------------+-------------+------------+
             | 41   42   43 |
             | 44 bottom 45 |
             | 46   47   48 |
             +--------------+
```

```
sage: rubik
The Rubik's cube group with generators R,L,F,B,U,D in SymmetricGroup(48).
```

```
>>> from sage.all import *
>>> rubik
The Rubik's cube group with generators R,L,F,B,U,D in SymmetricGroup(48).
```

B()

Return the generator B in Singmaster notation.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.B()
(1,14,48,27)(2,12,47,29)(3,9,46,32)(33,35,40,38)(34,37,39,36)
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> rubik.B()
(1,14,48,27)(2,12,47,29)(3,9,46,32)(33,35,40,38)(34,37,39,36)
```

D()

Return the generator D in Singmaster notation.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.D()
(14,22,30,38)(15,23,31,39)(16,24,32,40)(41,43,48,46)(42,45,47,44)
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> rubik.D()
(14,22,30,38)(15,23,31,39)(16,24,32,40)(41,43,48,46)(42,45,47,44)
```

F()

Return the generator F in Singmaster notation.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.F()
(6,25,43,16)(7,28,42,13)(8,30,41,11)(17,19,24,22)(18,21,23,20)
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> rubik.F()
(6,25,43,16)(7,28,42,13)(8,30,41,11)(17,19,24,22)(18,21,23,20)
```

L()

Return the generator L in Singmaster notation.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.L()
(1,17,41,40)(4,20,44,37)(6,22,46,35)(9,11,16,14)(10,13,15,12)
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> rubik.L()
(1,17,41,40)(4,20,44,37)(6,22,46,35)(9,11,16,14)(10,13,15,12)
```

R()

Return the generator R in Singmaster notation.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.R()
(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> rubik.R()
(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)
```

U()

Return the generator U in Singmaster notation.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.U()
(1,3,8,6)(2,5,7,4)(9,33,25,17)(10,34,26,18)(11,35,27,19)
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> rubik.U()
(1,3,8,6)(2,5,7,4)(9,33,25,17)(10,34,26,18)(11,35,27,19)
```

display2d(*mv*)

Print the 2d representation of `self`.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.display2d("R")
             +--------------+
             |  1    2   38 |
             |  4   top  36 |
             |  6    7   33 |
+------------+--------------+--------------+------------+
|  9  10  11 | 17   18    3 | 27   29   32 | 48  34  35 |
| 12 left 13 | 20  front  5 | 26 right  31 | 45 rear 37 |
| 14  15  16 | 22   23    8 | 25   28   30 | 43  39  40 |
+------------+--------------+--------------+------------+
             | 41   42   19 |
             | 44 bottom 21 |
             | 46   47   24 |
             +--------------+
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> rubik.display2d("R")
             +--------------+
             |  1    2   38 |
             |  4   top  36 |
             |  6    7   33 |
+------------+--------------+--------------+------------+
|  9  10  11 | 17   18    3 | 27   29   32 | 48  34  35 |
| 12 left 13 | 20  front  5 | 26 right  31 | 45 rear 37 |
| 14  15  16 | 22   23    8 | 25   28   30 | 43  39  40 |
+------------+--------------+--------------+------------+
             | 41   42   19 |
             | 44 bottom 21 |
             | 46   47   24 |
             +--------------+
```

faces(*mv*)

Return the dictionary of faces created by the effect of the move mv, which is a string of the form $X^a * Y^b * ...$, where $X, Y, \ldots$ are in $\{R, L, F, B, U, D\}$ and $a, b, \ldots$ are integers. We call this ordering of the faces the "BDFLRU, L2R, T2B ordering".

EXAMPLES:

```
sage: rubik = CubeGroup()
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
```

Here is the dictionary of the solved state:

```
sage: sorted(rubik.faces("").items())
[('back', [[33, 34, 35], [36, 0, 37], [38, 39, 40]]),
```

(continues on next page)

(continued from previous page)

```
 ('down', [[41, 42, 43], [44, 0, 45], [46, 47, 48]]),
 ('front', [[17, 18, 19], [20, 0, 21], [22, 23, 24]]),
 ('left', [[9, 10, 11], [12, 0, 13], [14, 15, 16]]),
 ('right', [[25, 26, 27], [28, 0, 29], [30, 31, 32]]),
 ('up', [[1, 2, 3], [4, 0, 5], [6, 7, 8]])]
```

```
>>> from sage.all import *
>>> sorted(rubik.faces("").items())
[('back', [[33, 34, 35], [36, 0, 37], [38, 39, 40]]),
 ('down', [[41, 42, 43], [44, 0, 45], [46, 47, 48]]),
 ('front', [[17, 18, 19], [20, 0, 21], [22, 23, 24]]),
 ('left', [[9, 10, 11], [12, 0, 13], [14, 15, 16]]),
 ('right', [[25, 26, 27], [28, 0, 29], [30, 31, 32]]),
 ('up', [[1, 2, 3], [4, 0, 5], [6, 7, 8]])]
```

Now the dictionary of the state obtained after making the move R followed by L:

```
sage: sorted(rubik.faces("R*U").items())
[('back', [[48, 26, 27], [45, 0, 37], [43, 39, 40]]),
 ('down', [[41, 42, 11], [44, 0, 21], [46, 47, 24]]),
 ('front', [[9, 10, 8], [20, 0, 7], [22, 23, 6]]),
 ('left', [[33, 34, 35], [12, 0, 13], [14, 15, 16]]),
 ('right', [[19, 29, 32], [18, 0, 31], [17, 28, 30]]),
 ('up', [[3, 5, 38], [2, 0, 36], [1, 4, 25]])]
```

```
>>> from sage.all import *
>>> sorted(rubik.faces("R*U").items())
[('back', [[48, 26, 27], [45, 0, 37], [43, 39, 40]]),
 ('down', [[41, 42, 11], [44, 0, 21], [46, 47, 24]]),
 ('front', [[9, 10, 8], [20, 0, 7], [22, 23, 6]]),
 ('left', [[33, 34, 35], [12, 0, 13], [14, 15, 16]]),
 ('right', [[19, 29, 32], [18, 0, 31], [17, 28, 30]]),
 ('up', [[3, 5, 38], [2, 0, 36], [1, 4, 25]])]
```

facets(*g=None*)

Return the set of facets on which the group acts. This function is a "constant".

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.facets() == list(range(1,49))
True
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> rubik.facets() == list(range(Integer(1),Integer(49)))
True
```

gen_names()

Return the names of the generators.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.gen_names()
['B', 'D', 'F', 'L', 'R', 'U']
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> rubik.gen_names()
['B', 'D', 'F', 'L', 'R', 'U']
```

legal(*state*, *mode='quiet'*)

Return 1 (true) if the dictionary `state` (in the same format as returned by the faces method) represents a legal position (or state) of the Rubik's cube or 0 (false) otherwise.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: r0 = rubik.faces("")
sage: r1 = {'back': [[33, 34, 35], [36, 0, 37], [38, 39, 40]], 'down': [[41,␣
→42, 43], [44, 0, 45], [46, 47, 48]],'front': [[17, 18, 19], [20, 0, 21],␣
→[22, 23, 24]],'left': [[9, 10, 11], [12, 0, 13], [14, 15, 16]],'right':␣
→[[25, 26, 27], [28, 0, 29], [30, 31, 32]],'up': [[1, 2, 3], [4, 0, 5], [6,␣
→8, 7]]}
sage: rubik.legal(r0)
1
sage: rubik.legal(r0,"verbose")
(1, ())
sage: rubik.legal(r1)
0
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> r0 = rubik.faces("")
>>> r1 = {'back': [[Integer(33), Integer(34), Integer(35)], [Integer(36),␣
→Integer(0), Integer(37)], [Integer(38), Integer(39), Integer(40)]], 'down':␣
→[[Integer(41), Integer(42), Integer(43)], [Integer(44), Integer(0),␣
→Integer(45)], [Integer(46), Integer(47), Integer(48)]],'front':␣
→[[Integer(17), Integer(18), Integer(19)], [Integer(20), Integer(0),␣
→Integer(21)], [Integer(22), Integer(23), Integer(24)]],'left': [[Integer(9),
→ Integer(10), Integer(11)], [Integer(12), Integer(0), Integer(13)],␣
→[Integer(14), Integer(15), Integer(16)]],'right': [[Integer(25),␣
→Integer(26), Integer(27)], [Integer(28), Integer(0), Integer(29)],␣
→[Integer(30), Integer(31), Integer(32)]],'up': [[Integer(1), Integer(2),␣
→Integer(3)], [Integer(4), Integer(0), Integer(5)], [Integer(6), Integer(8),␣
→Integer(7)]]}
>>> rubik.legal(r0)
1
>>> rubik.legal(r0,"verbose")
(1, ())
>>> rubik.legal(r1)
0
```

move(*mv*)

Return the group element and the reordered list of facets, as moved by the list `mv` (read left-to-right)

INPUT:

- mv – string of the form `Xa*Yb*...`, where `X`, `Y`, … are in `R`, `L`, `F`, `B`, `U`, `D` and `a`, `b`, … are integers

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.move("")[0]
()
sage: rubik.move("R")[0]
(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)
sage: rubik.R()
(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> rubik.move("")[Integer(0)]
()
>>> rubik.move("R")[Integer(0)]
(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)
>>> rubik.R()
(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)
```

parse(*mv*, *check=True*)

This function allows one to create the permutation group element from a variety of formats.

INPUT:

- `mv` – can one of the following:
  - `list` – list of facets (as returned by self.facets())
  - `dict` – list of faces (as returned by `self.faces()`)
  - `str` – either cycle notation (passed to GAP) or a product of generators or Singmaster notation
  - `perm_group element` – returned as an element of `self`
- `check` – check if the input is valid

EXAMPLES:

```
sage: C = CubeGroup()
sage: C.parse(list(range(1,49)))
()
sage: g = C.parse("L"); g
(1,17,41,40)(4,20,44,37)(6,22,46,35)(9,11,16,14)(10,13,15,12)
sage: C.parse(str(g)) == g
True
sage: facets = C.facets(g); facets
[17, 2, 3, 20, 5, 22, 7, 8, 11, 13, 16, 10, 15, 9, 12, 14, 41, 18, 19, 44, 21,
↪ 46, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 6, 36, 4, 38, 39, 1,␣
↪40, 42, 43, 37, 45, 35, 47, 48]
sage: C.parse(facets)
(1,17,41,40)(4,20,44,37)(6,22,46,35)(9,11,16,14)(10,13,15,12)
sage: C.parse(facets) == g
True
sage: faces = C.faces("L"); faces
```

(continues on next page)

(continued from previous page)

```
{'back': [[33, 34, 6], [36, 0, 4], [38, 39, 1]],
 'down': [[40, 42, 43], [37, 0, 45], [35, 47, 48]],
 'front': [[41, 18, 19], [44, 0, 21], [46, 23, 24]],
 'left': [[11, 13, 16], [10, 0, 15], [9, 12, 14]],
 'right': [[25, 26, 27], [28, 0, 29], [30, 31, 32]],
 'up': [[17, 2, 3], [20, 0, 5], [22, 7, 8]]}
sage: C.parse(faces) == C.parse("L")
True
sage: C.parse("L' R2") == C.parse("L^(-1)*R^2")
True
sage: C.parse("L' R2")
(1,40,41,17)(3,43)(4,37,44,20)(5,45)(6,35,46,22)(8,48)(9,14,16,11)(10,12,15,
↪13)(19,38)(21,36)(24,33)(25,32)(26,31)(27,30)(28,29)
sage: C.parse("L^4")
()
sage: C.parse("L^(-1)*R")
(1,40,41,17)(3,38,43,19)(4,37,44,20)(5,36,45,21)(6,35,46,22)(8,33,48,24)(9,14,
↪16,11)(10,12,15,13)(25,27,32,30)(26,29,31,28)
```

```
>>> from sage.all import *
>>> C = CubeGroup()
>>> C.parse(list(range(Integer(1),Integer(49))))
()
>>> g = C.parse("L"); g
(1,17,41,40)(4,20,44,37)(6,22,46,35)(9,11,16,14)(10,13,15,12)
>>> C.parse(str(g)) == g
True
>>> facets = C.facets(g); facets
[17, 2, 3, 20, 5, 22, 7, 8, 11, 13, 16, 10, 15, 9, 12, 14, 41, 18, 19, 44, 21,
↪ 46, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 6, 36, 4, 38, 39, 1,␣
↪40, 42, 43, 37, 45, 35, 47, 48]
>>> C.parse(facets)
(1,17,41,40)(4,20,44,37)(6,22,46,35)(9,11,16,14)(10,13,15,12)
>>> C.parse(facets) == g
True
>>> faces = C.faces("L"); faces
{'back': [[33, 34, 6], [36, 0, 4], [38, 39, 1]],
 'down': [[40, 42, 43], [37, 0, 45], [35, 47, 48]],
 'front': [[41, 18, 19], [44, 0, 21], [46, 23, 24]],
 'left': [[11, 13, 16], [10, 0, 15], [9, 12, 14]],
 'right': [[25, 26, 27], [28, 0, 29], [30, 31, 32]],
 'up': [[17, 2, 3], [20, 0, 5], [22, 7, 8]]}
>>> C.parse(faces) == C.parse("L")
True
>>> C.parse("L' R2") == C.parse("L^(-1)*R^2")
True
>>> C.parse("L' R2")
(1,40,41,17)(3,43)(4,37,44,20)(5,45)(6,35,46,22)(8,48)(9,14,16,11)(10,12,15,
↪13)(19,38)(21,36)(24,33)(25,32)(26,31)(27,30)(28,29)
>>> C.parse("L^4")
()
>>> C.parse("L^(-1)*R")
```

(continues on next page)

(continued from previous page)

```
(1,40,41,17)(3,38,43,19)(4,37,44,20)(5,36,45,21)(6,35,46,22)(8,33,48,24)(9,14,
↪16,11)(10,12,15,13)(25,27,32,30)(26,29,31,28)
```

plot3d_cube(*mv*, *title=True*)

Displays F, U, R faces of the cube after the given move `mv`. Mostly included for the purpose of drawing pictures and checking moves.

INPUT:

- `mv` – string in the Singmaster notation
- `title` – boolean (default: `True`); display the title information

The first one below is "superflip+4 spot" (in 26q* moves) and the second one is the superflip (in 20f* moves). Type show(P) to view them.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: P = rubik.plot3d_cube("U^2*F*U^2*L*R^(-1)*F^2*U*F^3*B^3*R*L*U^2*R*D^
↪3*U*L^3*R*D*R^3*L^3*D^2")          # needs sage.plot
sage: P = rubik.plot3d_cube("R*L*D^2*B^3*L^2*F^2*R^2*U^3*D*R^3*D^2*F^3*B^3*D^
↪3*F^2*D^3*R^2*U^3*F^2*D^3")     # needs sage.plot
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> P = rubik.plot3d_cube("U^2*F*U^2*L*R^(-1)*F^2*U*F^3*B^3*R*L*U^2*R*D^3*U*L^
↪3*R*D*R^3*L^3*D^2")          # needs sage.plot
>>> P = rubik.plot3d_cube("R*L*D^2*B^3*L^2*F^2*R^2*U^3*D*R^3*D^2*F^3*B^3*D^
↪3*F^2*D^3*R^2*U^3*F^2*D^3")     # needs sage.plot
```

plot_cube(*mv*, *title=True*, *colors=[(1, 0.63, 1), (1, 1, 0), (1, 0, 0), (0, 1, 0), (1, 0.6, 0.3), (0, 0, 1)]*)

Input the move mv, as a string in the Singmaster notation, and output the 2D plot of the cube in that state.

Type `P.show()` to display any of the plots below.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: P = rubik.plot_cube("R^2*U^2*R^2*U^2*R^2*U^2", title=False)          #␣
↪needs sage.plot
sage: # (R^2U^2)^3  permutes 2 pairs of edges (uf,ub)(fr,br)
sage: P = rubik.plot_cube("R*L*D^2*B^3*L^2*F^2*R^2*U^3*D*R^3*D^2*F^3*B^3*D^
↪3*F^2*D^3*R^2*U^3*F^2*D^3")       # needs sage.plot
sage: # the superflip (in 20f* moves)
sage: P = rubik.plot_cube("U^2*F*U^2*L*R^(-1)*F^2*U*F^3*B^3*R*L*U^2*R*D^3*U*L^
↪3*R*D*R^3*L^3*D^2")            # needs sage.plot
sage: # "superflip+4 spot" (in 26q* moves)
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> P = rubik.plot_cube("R^2*U^2*R^2*U^2*R^2*U^2", title=False)          #␣
↪needs sage.plot
>>> # (R^2U^2)^3  permutes 2 pairs of edges (uf,ub)(fr,br)
>>> P = rubik.plot_cube("R*L*D^2*B^3*L^2*F^2*R^2*U^3*D*R^3*D^2*F^3*B^3*D^3*F^
```

(continues on next page)

(continued from previous page)

```
↪2*D^3*R^2*U^3*F^2*D^3")       # needs sage.plot
>>> # the superflip (in 20f* moves)
>>> P = rubik.plot_cube("U^2*F*U^2*L*R^(-1)*F^2*U*F^3*B^3*R*L*U^2*R*D^3*U*L^
↪3*R*D*R^3*L^3*D^2")            # needs sage.plot
>>> # "superflip+4 spot" (in 26q* moves)
```

repr2d(*mv*)

Displays a 2D map of the Rubik's cube after the move mv has been made. Nothing is returned.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: print(rubik.repr2d(""))
                 ┌──────────────┐
                 │  1    2    3 │
                 │  4   top   5 │
                 │  6    7    8 │
    ┌────────────┼──────────────┼─────────────┬────────────┐
    │  9  10  11 │ 17   18   19 │ 25   26  27 │ 33  34  35 │
    │ 12 left 13 │ 20  front 21 │ 28 right 29 │ 36 rear 37 │
    │ 14  15  16 │ 22   23   24 │ 30   31  32 │ 38  39  40 │
    └────────────┼──────────────┼─────────────┴────────────┘
                 │ 41   42   43 │
                 │ 44 bottom 45 │
                 │ 46   47   48 │
                 └──────────────┘
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> print(rubik.repr2d(""))
                 ┌──────────────┐
                 │  1    2    3 │
                 │  4   top   5 │
                 │  6    7    8 │
    ┌────────────┼──────────────┼─────────────┬────────────┐
    │  9  10  11 │ 17   18   19 │ 25   26  27 │ 33  34  35 │
    │ 12 left 13 │ 20  front 21 │ 28 right 29 │ 36 rear 37 │
    │ 14  15  16 │ 22   23   24 │ 30   31  32 │ 38  39  40 │
    └────────────┼──────────────┼─────────────┴────────────┘
                 │ 41   42   43 │
                 │ 44 bottom 45 │
                 │ 46   47   48 │
                 └──────────────┘
```

```
sage: print(rubik.repr2d("R"))
                 ┌──────────────┐
                 │  1    2   38 │
                 │  4   top  36 │
                 │  6    7   33 │
    ┌────────────┼──────────────┼─────────────┬────────────┐
    │  9  10  11 │ 17   18    3 │ 27   29  32 │ 48  34  35 │
    │ 12 left 13 │ 20  front  5 │ 26 right 31 │ 45 rear 37 │
```

(continues on next page)

(continued from previous page)

```
│ 14  15  16 │ 22   23    8 │ 25   28  30 │ 43  39  40 │
└────────────┼──────────────┼─────────────┴────────────┘
             │ 41   42   19 │
             │ 44 bottom 21 │
             │ 46   47   24 │
             └──────────────┘
```

```
>>> from sage.all import *
>>> print(rubik.repr2d("R"))
             ┌──────────────┐
             │  1    2   38 │
             │  4   top  36 │
             │  6    7   33 │
┌────────────┼──────────────┼─────────────┬────────────┐
│  9  10  11 │ 17   18    3 │ 27   29  32 │ 48  34  35 │
│ 12 left 13 │ 20  front  5 │ 26 right 31 │ 45 rear 37 │
│ 14  15  16 │ 22   23    8 │ 25   28  30 │ 43  39  40 │
└────────────┼──────────────┼─────────────┴────────────┘
             │ 41   42   19 │
             │ 44 bottom 21 │
             │ 46   47   24 │
             └──────────────┘
```

You can see the right face has been rotated but not the left face.

solve(*state*, *algorithm='default'*)

Solve the cube in the `state`, given as a dictionary as in `legal`. See the `solve` method of the RubiksCube class for more details.

This may use GAP's `EpimorphismFromFreeGroup` and `PreImagesRepresentative` as explained below, if 'gap' is passed in as the algorithm.

This algorithm

1. constructs the free group on 6 generators then computes a reasonable set of relations which they satisfy
2. computes a homomorphism from the cube group to this free group quotient
3. takes the cube position, regarded as a group element, and maps it over to the free group quotient
4. using those relations and tricks from combinatorial group theory (stabilizer chains), solves the "word problem" for that element.
5. uses python string parsing to rewrite that in cube notation.

The Rubik's cube group has about 4.3×10^{19} elements, so this process is time-consuming. See https://www.gap-system.org/Doc/Examples/rubik.html for an interesting discussion of some GAP code analyzing the Rubik's cube.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: state = rubik.faces("R")
sage: rubik.solve(state)
'R'
sage: state = rubik.faces("R*U")
```

(continues on next page)

(continued from previous page)

```
sage: rubik.solve(state, algorithm='gap')        # long time
'R*U'
```

```
>>> from sage.all import *
>>> rubik = CubeGroup()
>>> state = rubik.faces("R")
>>> rubik.solve(state)
'R'
>>> state = rubik.faces("R*U")
>>> rubik.solve(state, algorithm='gap')        # long time
'R*U'
```

You can also check this another (but similar) way using the `word_problem` method (eg, G = rubik.group(); g = G("(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)"); g.word_problem([b,d,f,l,r,u]), though the output will be less intuitive).

class `sage.groups.perm_gps.cubegroup.`**`RubiksCube`**(*state=None*, *history=[]*, *colors=[(1, 0.63, 1), (1, 1, 0), (1, 0, 0), (0, 1, 0), (1, 0.6, 0.3), (0, 0, 1)]*)

Bases: `SageObject`

The Rubik's cube (in a given state).

EXAMPLES:

```
sage: C = RubiksCube().move("R U R'")
sage: C.show3d()                                                    #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> C = RubiksCube().move("R U R'")
>>> C.show3d()                                                    #␣
↪needs sage.plot
```

```
sage: C = RubiksCube("R*L"); C
              ┌──────────────┐
              │ 17    2   38 │
              │ 20   top  36 │
              │ 22    7   33 │
┌─────────────┼──────────────┼──────────────┬─────────────┐
│ 11   13  16 │ 41   18    3 │ 27   29   32 │ 48  34    6 │
│ 10 left  15 │ 44  front  5 │ 26 right  31 │ 45 rear   4 │
│  9   12  14 │ 46   23    8 │ 25   28   30 │ 43  39    1 │
└─────────────┼──────────────┼──────────────┴─────────────┘
              │ 40   42   19 │
              │ 37 bottom 21 │
              │ 35   47   24 │
              └──────────────┘
sage: C.show()                                                      #␣
↪needs sage.plot
sage: C.solve(algorithm='gap')  # long time
'L*R'
sage: C == RubiksCube("L*R")
True
```

```
>>> from sage.all import *
>>> C = RubiksCube("R*L"); C
             ┌──────────────┐
             │ 17    2   38 │
             │ 20   top  36 │
             │ 22    7   33 │
┌────────────┼──────────────┼─────────────┬────────────┐
│ 11  13  16 │ 41   18    3 │ 27   29  32 │ 48  34   6 │
│ 10 left 15 │ 44  front  5 │ 26 right 31 │ 45 rear  4 │
│  9  12  14 │ 46   23    8 │ 25   28  30 │ 43  39   1 │
└────────────┼──────────────┼─────────────┴────────────┘
             │ 40   42   19 │
             │ 37 bottom 21 │
             │ 35   47   24 │
             └──────────────┘
>>> C.show()                                                        #␣
↪needs sage.plot
>>> C.solve(algorithm='gap')  # long time
'L*R'
>>> C == RubiksCube("L*R")
True
```

cubie(*size*, *gap*, *x*, *y*, *z*, *colors*, *stickers=True*)

Return the cubie at (x, y, z).

INPUT:

- `size` – the size of the cubie
- `gap` – the gap between cubies
- `x`, `y`, `z` – the position of the cubie
- `colors` – the list of colors
- `stickers` – boolean (default: `True`); whether to display stickers

EXAMPLES:

```
sage: C = RubiksCube("R*U")
sage: C.cubie(0.15, 0.025, 0,0,0, C.colors*3)                       #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> C = RubiksCube("R*U")
>>> C.cubie(RealNumber('0.15'), RealNumber('0.025'), Integer(0),Integer(0),
↪Integer(0), C.colors*Integer(3))                          # needs sage.
↪plot
Graphics3d Object
```

facets()

Return the facets of `self`.

EXAMPLES:

```
sage: C = RubiksCube("R*U")
sage: C.facets()
[3, 5, 38, 2, 36, 1, 4, 25, 33, 34, 35, 12, 13, 14, 15, 16, 9, 10,
 8, 20, 7, 22, 23, 6, 19, 29, 32, 18, 31, 17, 28, 30, 48, 26, 27,
 45, 37, 43, 39, 40, 41, 42, 11, 44, 21, 46, 47, 24]
```

```
>>> from sage.all import *
>>> C = RubiksCube("R*U")
>>> C.facets()
[3, 5, 38, 2, 36, 1, 4, 25, 33, 34, 35, 12, 13, 14, 15, 16, 9, 10,
 8, 20, 7, 22, 23, 6, 19, 29, 32, 18, 31, 17, 28, 30, 48, 26, 27,
 45, 37, 43, 39, 40, 41, 42, 11, 44, 21, 46, 47, 24]
```

move (*g*)

Move the Rubik's cube by `g`.

EXAMPLES:

```
sage: RubiksCube().move("R*U") == RubiksCube("R*U")
True
```

```
>>> from sage.all import *
>>> RubiksCube().move("R*U") == RubiksCube("R*U")
True
```

plot ()

Return a plot of `self`.

EXAMPLES:

```
sage: C = RubiksCube("R*U")
sage: C.plot()                                                          #␣
↪needs sage.plot
Graphics object consisting of 55 graphics primitives
```

```
>>> from sage.all import *
>>> C = RubiksCube("R*U")
>>> C.plot()                                                        #␣
↪needs sage.plot
Graphics object consisting of 55 graphics primitives
```

plot3d (*stickers=True*)

Return a 3D plot of `self`.

EXAMPLES:

```
sage: C = RubiksCube("R*U")
sage: C.plot3d()                                                        #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> C = RubiksCube("R*U")
```

(continues on next page)

(continued from previous page)

```
>>> C.plot3d()                                                              #␣
↪needs sage.plot
Graphics3d Object
```

scramble(*moves=30*)

Scramble the Rubik's cube.

EXAMPLES:

```
sage: C = RubiksCube()
sage: C.scramble() # random
                 ┌──────────────┐
                 │  3    5   38 │
                 │  2   top  36 │
                 │  1    4   25 │
 ┌───────────────┼──────────────┼───────────────┬──────────────┐
 │ 33   34   35  │  9   10    8 │ 19   29   32  │ 48  26   27  │
 │ 12  left  13  │ 20  front  7 │ 18  right 31  │ 45  rear 37  │
 │ 14   15   16  │ 22   23    6 │ 17   28   30  │ 43  39   40  │
 └───────────────┼──────────────┼───────────────┴──────────────┘
                 │ 41   42   11 │
                 │ 44 bottom 21 │
                 │ 46   47   24 │
                 └──────────────┘
```

```
>>> from sage.all import *
>>> C = RubiksCube()
>>> C.scramble() # random
                 ┌──────────────┐
                 │  3    5   38 │
                 │  2   top  36 │
                 │  1    4   25 │
 ┌───────────────┼──────────────┼───────────────┬──────────────┐
 │ 33   34   35  │  9   10    8 │ 19   29   32  │ 48  26   27  │
 │ 12  left  13  │ 20  front  7 │ 18  right 31  │ 45  rear 37  │
 │ 14   15   16  │ 22   23    6 │ 17   28   30  │ 43  39   40  │
 └───────────────┼──────────────┼───────────────┴──────────────┘
                 │ 41   42   11 │
                 │ 44 bottom 21 │
                 │ 46   47   24 │
                 └──────────────┘
```

show()

Show a plot of `self`.

EXAMPLES:

```
sage: C = RubiksCube("R*U")
sage: C.show()                                                                #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> C = RubiksCube("R*U")
```

(continues on next page)

(continued from previous page)

```
>>> C.show()                                                                #␣
↪needs sage.plot
```

show3d()

Show a 3D plot of `self`.

EXAMPLES:

```
sage: C = RubiksCube("R*U")
sage: C.show3d()                                                            #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> C = RubiksCube("R*U")
>>> C.show3d()                                                              #␣
↪needs sage.plot
```

solve(*algorithm='default'*, *timeout=15*)

Solve the Rubik's cube.

INPUT:

- `algorithm` – must be one of the following:
  - `hybrid` – (default) try `kociemba` for timeout seconds, then `dietz`
  - `kociemba` – use Dik T. Winter's program (reasonable speed, few moves)
  - `dietz` – use Eric Dietz's cubex program (fast but lots of moves)
  - `optimal` – use Michael Reid's optimal program (may take a long time)
  - `gap` – use GAP word solution (can be slow)

Any choice other than `gap` requires the optional package `rubiks`. If the package is not installed, the `gap` algorithm is used by default.

EXAMPLES:

```
sage: C = RubiksCube("R U F L B D")
sage: C.solve()              # optional - rubiks
'R U F L B D'
```

```
>>> from sage.all import *
>>> C = RubiksCube("R U F L B D")
>>> C.solve()              # optional - rubiks
'R U F L B D'
```

Dietz's program is much faster, but may give highly non-optimal solutions:

```
sage: s = C.solve('dietz'); s   # optional - rubiks
"U' L' L' U L U' L U D L L D' L' D L' D' L D L' U' L D' L' U L' B'
 U' L' U B L D L D' U' L' U L B L B' L' U L U' L' F' L' F L' F L F'
 L' D' L' D D L D' B L B' L B' L B F' L F F B' L F' B D' D' L D B'
 B' L' D' B U' U' L' B' D' F' F' L D F'"
sage: C2 = RubiksCube(s)  # optional - rubiks
```

(continues on next page)

(continued from previous page)

```
sage: C == C2               # optional - rubiks
True
```

```
>>> from sage.all import *
>>> s = C.solve('dietz'); s   # optional - rubiks
"U' L' L' U L U' L U D L L D' L' D L' D' L D L' U' L D' L' U L' B'
 U' L' U B L D L D' U' L' U L B L B' L' U L U' L' F' L' F L' F L F'
 L' D' L' D D L D' B L B' L B' L B F' L F F B' L F' B D' D' L D B'
 B' L' D' B U' U' L' B' D' F' F' L D F'"
>>> C2 = RubiksCube(s)  # optional - rubiks
>>> C == C2             # optional - rubiks
True
```

undo()

Undo the last move of the Rubik's cube.

EXAMPLES:

```
sage: C = RubiksCube()
sage: D = C.move("R*U")
sage: D.undo() == C
True
```

```
>>> from sage.all import *
>>> C = RubiksCube()
>>> D = C.move("R*U")
>>> D.undo() == C
True
```

sage.groups.perm_gps.cubegroup.**color_of_square**(*facet*, *colors=['lpurple', 'yellow', 'red', 'green', 'orange', 'blue']*)

Return the color the facet has in the solved state.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import color_of_square
sage: color_of_square(41)
'blue'
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.cubegroup import color_of_square
>>> color_of_square(Integer(41))
'blue'
```

sage.groups.perm_gps.cubegroup.**create_poly**(*face*, *color*)

Create the polygon given by `face` with color `color`.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import create_poly, red
sage: create_poly('ur', red)                                                     #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.cubegroup import create_poly, red
>>> create_poly('ur', red)                                              #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

sage.groups.perm_gps.cubegroup.**cubie_centers**(*label*)

Return the cubie center list element given by `label`.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import cubie_centers
sage: cubie_centers(3)
[0, 2, 2]
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.cubegroup import cubie_centers
>>> cubie_centers(Integer(3))
[0, 2, 2]
```

sage.groups.perm_gps.cubegroup.**cubie_colors**(*label*, *state0*)

Return the color of the cubie given by `label` at `state0`.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import cubie_colors
sage: G = CubeGroup()
sage: g = G.parse("R*U")
sage: cubie_colors(3, G.facets(g))
[(1, 1, 1), (1, 0.63, 1), (1, 0.6, 0.3)]
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.cubegroup import cubie_colors
>>> G = CubeGroup()
>>> g = G.parse("R*U")
>>> cubie_colors(Integer(3), G.facets(g))
[(1, 1, 1), (1, 0.63, 1), (1, 0.6, 0.3)]
```

sage.groups.perm_gps.cubegroup.**cubie_faces**()

This provides a map from the 6 faces of the 27 cubies to the 48 facets of the larger cube.

-1,-1,-1 is left, top, front

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import cubie_faces
sage: sorted(cubie_faces().items())
[((-1, -1, -1), [6, 17, 11, 0, 0, 0]),
 ((-1, -1, 0), [4, 0, 10, 0, 0, 0]),
 ((-1, -1, 1), [1, 0, 9, 0, 35, 0]),
 ((-1, 0, -1), [0, 20, 13, 0, 0, 0]),
 ((-1, 0, 0), [0, 0, -5, 0, 0, 0]),
 ((-1, 0, 1), [0, 0, 12, 0, 37, 0]),
 ((-1, 1, -1), [0, 22, 16, 41, 0, 0]),
```

(continues on next page)

(continued from previous page)

```
 ((-1, 1, 0), [0, 0, 15, 44, 0, 0]),
 ((-1, 1, 1), [0, 0, 14, 46, 40, 0]),
 ((0, -1, -1), [7, 18, 0, 0, 0, 0]),
 ((0, -1, 0), [-6, 0, 0, 0, 0, 0]),
 ((0, -1, 1), [2, 0, 0, 0, 34, 0]),
 ((0, 0, -1), [0, -4, 0, 0, 0, 0]),
 ((0, 0, 0), [0, 0, 0, 0, 0, 0]),
 ((0, 0, 1), [0, 0, 0, 0, -2, 0]),
 ((0, 1, -1), [0, 23, 0, 42, 0, 0]),
 ((0, 1, 0), [0, 0, 0, -1, 0, 0]),
 ((0, 1, 1), [0, 0, 0, 47, 39, 0]),
 ((1, -1, -1), [8, 19, 0, 0, 0, 25]),
 ((1, -1, 0), [5, 0, 0, 0, 0, 26]),
 ((1, -1, 1), [3, 0, 0, 0, 33, 27]),
 ((1, 0, -1), [0, 21, 0, 0, 0, 28]),
 ((1, 0, 0), [0, 0, 0, 0, 0, -3]),
 ((1, 0, 1), [0, 0, 0, 0, 36, 29]),
 ((1, 1, -1), [0, 24, 0, 43, 0, 30]),
 ((1, 1, 0), [0, 0, 0, 45, 0, 31]),
 ((1, 1, 1), [0, 0, 0, 48, 38, 32])]
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.cubegroup import cubie_faces
>>> sorted(cubie_faces().items())
[((-1, -1, -1), [6, 17, 11, 0, 0, 0]),
 ((-1, -1, 0), [4, 0, 10, 0, 0, 0]),
 ((-1, -1, 1), [1, 0, 9, 0, 35, 0]),
 ((-1, 0, -1), [0, 20, 13, 0, 0, 0]),
 ((-1, 0, 0), [0, 0, -5, 0, 0, 0]),
 ((-1, 0, 1), [0, 0, 12, 0, 37, 0]),
 ((-1, 1, -1), [0, 22, 16, 41, 0, 0]),
 ((-1, 1, 0), [0, 0, 15, 44, 0, 0]),
 ((-1, 1, 1), [0, 0, 14, 46, 40, 0]),
 ((0, -1, -1), [7, 18, 0, 0, 0, 0]),
 ((0, -1, 0), [-6, 0, 0, 0, 0, 0]),
 ((0, -1, 1), [2, 0, 0, 0, 34, 0]),
 ((0, 0, -1), [0, -4, 0, 0, 0, 0]),
 ((0, 0, 0), [0, 0, 0, 0, 0, 0]),
 ((0, 0, 1), [0, 0, 0, 0, -2, 0]),
 ((0, 1, -1), [0, 23, 0, 42, 0, 0]),
 ((0, 1, 0), [0, 0, 0, -1, 0, 0]),
 ((0, 1, 1), [0, 0, 0, 47, 39, 0]),
 ((1, -1, -1), [8, 19, 0, 0, 0, 25]),
 ((1, -1, 0), [5, 0, 0, 0, 0, 26]),
 ((1, -1, 1), [3, 0, 0, 0, 33, 27]),
 ((1, 0, -1), [0, 21, 0, 0, 0, 28]),
 ((1, 0, 0), [0, 0, 0, 0, 0, -3]),
 ((1, 0, 1), [0, 0, 0, 0, 36, 29]),
 ((1, 1, -1), [0, 24, 0, 43, 0, 30]),
 ((1, 1, 0), [0, 0, 0, 45, 0, 31]),
 ((1, 1, 1), [0, 0, 0, 48, 38, 32])]
```

sage.groups.perm_gps.cubegroup.**index2singmaster**(*facet*)

Translate index used (eg, 43) to Singmaster facet notation (eg, fdr).

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import index2singmaster
sage: index2singmaster(41)
'dlf'
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.cubegroup import index2singmaster
>>> index2singmaster(Integer(41))
'dlf'
```

sage.groups.perm_gps.cubegroup.**inv_list**(*lst*)

Input a list of ints $1, \ldots, m$ (in any order), outputs inverse perm.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import inv_list
sage: L = [2,3,1]
sage: inv_list(L)
[3, 1, 2]
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.cubegroup import inv_list
>>> L = [Integer(2),Integer(3),Integer(1)]
>>> inv_list(L)
[3, 1, 2]
```

sage.groups.perm_gps.cubegroup.**plot3d_cubie**(*cnt*, *clrs*)

Plot the front, up and right face of a cubie centered at cnt and rgbcolors given by clrs (in the order FUR).

Type `P.show()` to view.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import plot3d_cubie, blue, red, green
sage: clrF = blue; clrU = red; clrR = green
sage: P = plot3d_cubie([1/2,1/2,1/2],[clrF,clrU,clrR])                    #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.cubegroup import plot3d_cubie, blue, red, green
>>> clrF = blue; clrU = red; clrR = green
>>> P = plot3d_cubie([Integer(1)/Integer(2),Integer(1)/Integer(2),Integer(1)/
↪Integer(2)],[clrF,clrU,clrR])                       # needs sage.plot
```

sage.groups.perm_gps.cubegroup.**polygon_plot3d**(*points*, *tilt=30*, *turn=30*, ***kwargs*)

Plot a polygon viewed from an angle determined by `tilt`, `turn`, and vertices `points`.

Warning

The ordering of the points is important to get "correct" and if you add several of these plots together, the one added first is also drawn first (ie, addition of Graphics objects is not commutative).

The following example produced a green-colored square with vertices at the points indicated.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import polygon_plot3d,green
sage: P = polygon_plot3d([[1,3,1],[2,3,1],[2,3,2],[1,3,2],[1,3,1]],          #␣
↪needs sage.plot
....:                    rgbcolor=green)
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.cubegroup import polygon_plot3d,green
>>> P = polygon_plot3d([[Integer(1),Integer(3),Integer(1)],[Integer(2),Integer(3),
↪Integer(1)],[Integer(2),Integer(3),Integer(2)],[Integer(1),Integer(3),
↪Integer(2)],[Integer(1),Integer(3),Integer(1)]],              # needs sage.plot
...                    rgbcolor=green)
```

sage.groups.perm_gps.cubegroup.**rotation_list**(*tilt*, *turn*)

Return a list $[\sin(\theta), \sin(\phi), \cos(\theta), \cos(\phi)]$ of rotations where θ is `tilt` and ϕ is `turn`.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import rotation_list
sage: rotation_list(30, 45)
[0.49999999999999994, 0.7071067811865475, 0.8660254037844387, 0.7071067811865476]
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.cubegroup import rotation_list
>>> rotation_list(Integer(30), Integer(45))
[0.49999999999999994, 0.7071067811865475, 0.8660254037844387, 0.7071067811865476]
```

sage.groups.perm_gps.cubegroup.**xproj**(*x*, *y*, *z*, *r*)

Return the x-projection of (x, y, z) rotated by r.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import rotation_list, xproj
sage: rot = rotation_list(30, 45)
sage: xproj(1,2,3,rot)
0.6123724356957945
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.cubegroup import rotation_list, xproj
>>> rot = rotation_list(Integer(30), Integer(45))
>>> xproj(Integer(1),Integer(2),Integer(3),rot)
0.6123724356957945
```

sage.groups.perm_gps.cubegroup.**yproj**(*x*, *y*, *z*, *r*)

Return the y-projection of (x, y, z) rotated by r.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import rotation_list, yproj
sage: rot = rotation_list(30, 45)
sage: yproj(1,2,3,rot)
1.378497416975604
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.cubegroup import rotation_list, yproj
>>> rot = rotation_list(Integer(30), Integer(45))
>>> yproj(Integer(1),Integer(2),Integer(3),rot)
1.378497416975604
```

27.8 Conjugacy Classes Of The Symmetric Group

AUTHORS:

- Vincent Delecroix, Travis Scrimshaw (2014-11-23)

class sage.groups.perm_gps.symgp_conjugacy_class.**PermutationsConjugacyClass**(*P*, *part*)

Bases: *SymmetricGroupConjugacyClassMixin*, *ConjugacyClass*

A conjugacy class of the permutations of n.

INPUT:

- P – the permutations of n
- part – a partition or an element of P

set()

The set of all elements in the conjugacy class self.

EXAMPLES:

```
sage: G = Permutations(3)
sage: g = G([2, 1, 3])
sage: C = G.conjugacy_class(g)                                                    #␣
↪needs sage.combinat
sage: S = [[1, 3, 2], [2, 1, 3], [3, 2, 1]]
sage: C.set() == Set(G(x) for x in S)                                             #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> G = Permutations(Integer(3))
>>> g = G([Integer(2), Integer(1), Integer(3)])
>>> C = G.conjugacy_class(g)                                                      #␣
↪needs sage.combinat
>>> S = [[Integer(1), Integer(3), Integer(2)], [Integer(2), Integer(1),␣
↪Integer(3)], [Integer(3), Integer(2), Integer(1)]]
>>> C.set() == Set(G(x) for x in S)                                               #␣
↪needs sage.combinat
True
```

class sage.groups.perm_gps.symgp_conjugacy_class.**SymmetricGroupConjugacyClass**(*group*, *part*)

Bases: *SymmetricGroupConjugacyClassMixin*, *ConjugacyClassGAP*

A conjugacy class of the symmetric group.

INPUT:

- group – the symmetric group
- part – a partition or an element of group

set()

The set of all elements in the conjugacy class self.

EXAMPLES:

```
sage: G = SymmetricGroup(3)
sage: g = G((1,2))
sage: C = G.conjugacy_class(g)                                          #␣
↪needs sage.combinat
sage: S = [(2,3), (1,2), (1,3)]
sage: C.set() == Set(G(x) for x in S)                                   #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> g = G((Integer(1),Integer(2)))
>>> C = G.conjugacy_class(g)                                          #␣
↪needs sage.combinat
>>> S = [(Integer(2),Integer(3)), (Integer(1),Integer(2)), (Integer(1),
↪Integer(3))]
>>> C.set() == Set(G(x) for x in S)                                   #␣
↪needs sage.combinat
True
```

class sage.groups.perm_gps.symgp_conjugacy_class.**SymmetricGroupConjugacyClassMixin**(*domain*, *part*)

Bases: object

Mixin class which contains methods for conjugacy classes of the symmetric group.

partition()

Return the partition of self.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: g = G([(1,2), (3,4,5)])
sage: C = G.conjugacy_class(g)                                          #␣
↪needs sage.combinat
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> g = G([(Integer(1),Integer(2)), (Integer(3),Integer(4),Integer(5))])
```

(continues on next page)

(continued from previous page)

```
>>> C = G.conjugacy_class(g)                                                #␣
↪needs sage.combinat
```

sage.groups.perm_gps.symgp_conjugacy_class.**conjugacy_class_iterator**(*part*, *S=None*)

Return an iterator over the conjugacy class associated to the partition part.

The elements are given as a list of tuples, each tuple being a cycle.

INPUT:

- part – partition
- S – (default: $\{1, 2, \ldots, n\}$, where n is the size of part) a set

OUTPUT: an iterator over the conjugacy class consisting of all permutations of the set S whose cycle type is part

EXAMPLES:

```
sage: from sage.groups.perm_gps.symgp_conjugacy_class import conjugacy_class_
↪iterator       # needs sage.combinat
sage: for p in conjugacy_class_iterator([2,2]): print(p)                    #␣
↪needs sage.combinat
[(1, 2), (3, 4)]
[(1, 4), (2, 3)]
[(1, 3), (2, 4)]
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.symgp_conjugacy_class import conjugacy_class_
↪iterator       # needs sage.combinat
>>> for p in conjugacy_class_iterator([Integer(2),Integer(2)]): print(p)     ␣
↪               # needs sage.combinat
[(1, 2), (3, 4)]
[(1, 4), (2, 3)]
[(1, 3), (2, 4)]
```

In order to get permutations, one just has to wrap:

```
sage: S = SymmetricGroup(5)
sage: for p in conjugacy_class_iterator([3,2]): print(S(p))                 #␣
↪needs sage.combinat
(1,3)(2,4,5)
(1,3)(2,5,4)
(1,2)(3,4,5)
(1,2)(3,5,4)
...
(1,4)(2,3,5)
(1,4)(2,5,3)
```

```
>>> from sage.all import *
>>> S = SymmetricGroup(Integer(5))
>>> for p in conjugacy_class_iterator([Integer(3),Integer(2)]): print(S(p))  ␣
↪               # needs sage.combinat
(1,3)(2,4,5)
(1,3)(2,5,4)
(1,2)(3,4,5)
```

(continues on next page)

(continued from previous page)

```
(1,2)(3,5,4)
...
(1,4)(2,3,5)
(1,4)(2,5,3)
```

Check that the number of elements is the number of elements in the conjugacy class:

```
sage: s = lambda p: sum(1 for _ in conjugacy_class_iterator(p))
sage: all(s(p) == p.conjugacy_class_size() for p in Partitions(5))              #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> s = lambda p: sum(Integer(1) for _ in conjugacy_class_iterator(p))
>>> all(s(p) == p.conjugacy_class_size() for p in Partitions(Integer(5)))       ␣
↪     # needs sage.combinat
True
```

It is also possible to specify any underlying set:

```
sage: it = conjugacy_class_iterator([2,2,2], 'abcdef')                          #␣
↪needs sage.combinat
sage: sorted(flatten(next(it)))                                                 #␣
↪needs sage.combinat
['a', 'b', 'c', 'd', 'e', 'f']
sage: all(len(x) == 2 for x in next(it))                                        #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> it = conjugacy_class_iterator([Integer(2),Integer(2),Integer(2)], 'abcdef')  ␣
↪                    # needs sage.combinat
>>> sorted(flatten(next(it)))                                                 #␣
↪needs sage.combinat
['a', 'b', 'c', 'd', 'e', 'f']
>>> all(len(x) == Integer(2) for x in next(it))                                 ␣
↪     # needs sage.combinat
True
```

sage.groups.perm_gps.symgp_conjugacy_class.**default_representative**(*part*, *G*)

Construct the default representative for the conjugacy class of cycle type `part` of a symmetric group `G`.

Let λ be a partition of n. We pick a representative by

$$(1, 2, \ldots, \lambda_1)(\lambda_1 + 1, \ldots, \lambda_1 + \lambda_2)(\lambda_1 + \lambda_2 + \cdots + \lambda_{\ell-1}, \ldots, n),$$

where ℓ is the length (or number of parts) of λ.

INPUT:

- `part` – partition
- `G` – a symmetric group

EXAMPLES:

```
sage: from sage.groups.perm_gps.symgp_conjugacy_class import default_
↪representative          # needs sage.combinat
sage: S = SymmetricGroup(4)
sage: for p in Partitions(4):                                                 #␣
↪needs sage.combinat
....:     print(default_representative(p, S))
(1,2,3,4)
(1,2,3)
(1,2)(3,4)
(1,2)
()
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.symgp_conjugacy_class import default_representative␣
↪        # needs sage.combinat
>>> S = SymmetricGroup(Integer(4))
>>> for p in Partitions(Integer(4)):                                            ␣
↪     # needs sage.combinat
...     print(default_representative(p, S))
(1,2,3,4)
(1,2,3)
(1,2)(3,4)
(1,2)
()
```

CHAPTER

TWENTYEIGHT

MATRIX AND AFFINE GROUPS

28.1 Library of Interesting Groups

Type `groups.matrix.<tab>` to access examples of groups implemented as permutation groups.

28.2 Base classes for Matrix Groups

AUTHORS:

- William Stein: initial version
- David Joyner (2006-03-15): degree, base_ring, _contains_, list, random, order methods; examples
- William Stein (2006-12): rewrite
- David Joyner (2007-12): Added invariant_generators (with Martin Albrecht and Simon King)
- David Joyner (2008-08): Added module_composition_factors (interface to GAP's MeatAxe implementation) and as_permutation_group (returns isomorphic PermutationGroup).
- Simon King (2010-05): Improve invariant_generators by using GAP for the construction of the Reynolds operator in Singular.
- Sebastian Oehms (2018-07): Add `subgroup()` and `ambient()` see Issue #25894

class sage.groups.matrix_gps.matrix_group.**MatrixGroup_base**

Bases: *Group*

Base class for all matrix groups.

This base class just holds the base ring, but not the degree. So it can be a base for affine groups where the natural matrix is larger than the degree of the affine group. Makes no assumption about the group except that its elements have a `matrix()` method.

ambient()

Return the ambient group of a subgroup.

OUTPUT:

A group containing `self`. If `self` has not been defined as a subgroup, we just return `self`.

EXAMPLES:

```
sage: G = GL(2, QQ)
sage: m = matrix(QQ, 2, 2, [[3, 0], [~5,1]])
sage: S = G.subgroup([m])
```

(continues on next page)

(continued from previous page)

```
sage: S.ambient() is G
True
```

```
>>> from sage.all import *
>>> G = GL(Integer(2), QQ)
>>> m = matrix(QQ, Integer(2), Integer(2), [[Integer(3), Integer(0)], [~
↪Integer(5),Integer(1)]])
>>> S = G.subgroup([m])
>>> S.ambient() is G
True
```

as_matrix_group()

Return a new matrix group from the generators.

This will throw away any extra structure (encoded in a derived class) that a group of special matrices has.

EXAMPLES:

```
sage: G = SU(4, GF(5))                                                    #␣
↪needs sage.rings.finite_rings
sage: G.as_matrix_group()                                                 #␣
↪needs sage.libs.gap sage.rings.finite_rings
Matrix group over Finite Field in a of size 5^2 with 2 generators (
[      a       0       0       0]  [      1       0 4*a + 3       0]
[      0 2*a + 3       0       0]  [      1       0       0       0]
[      0       0 4*a + 1       0]  [      0 2*a + 4       0       1]
[      0       0       0     3*a], [      0 3*a + 1       0       0]
)

sage: # needs sage.libs.gap
sage: G = GO(3, GF(5))
sage: G.as_matrix_group()
Matrix group over Finite Field of size 5 with 2 generators (
[2 0 0]  [0 1 0]
[0 3 0]  [1 4 4]
[0 0 1], [0 2 1]
)
```

```
>>> from sage.all import *
>>> G = SU(Integer(4), GF(Integer(5)))                                      ␣
↪              # needs sage.rings.finite_rings
>>> G.as_matrix_group()                                                   #␣
↪needs sage.libs.gap sage.rings.finite_rings
Matrix group over Finite Field in a of size 5^2 with 2 generators (
[      a       0       0       0]  [      1       0 4*a + 3       0]
[      0 2*a + 3       0       0]  [      1       0       0       0]
[      0       0 4*a + 1       0]  [      0 2*a + 4       0       1]
[      0       0       0     3*a], [      0 3*a + 1       0       0]
)

>>> # needs sage.libs.gap
>>> G = GO(Integer(3), GF(Integer(5)))
```

(continues on next page)

(continued from previous page)

```
>>> G.as_matrix_group()
Matrix group over Finite Field of size 5 with 2 generators (
[2 0 0]  [0 1 0]
[0 3 0]  [1 4 4]
[0 0 1], [0 2 1]
)
```

natural_representation(*base_ring=None*)

Return the natural representation of `self` over `base_ring`.

INPUT:

- `base_ring` – (optional) the base ring; the default is the base ring of `self`

EXAMPLES:

```
sage: G = groups.matrix.SL(6, 3)
sage: V = G.natural_representation()
sage: V
Natural representation of Special Linear Group of degree 6
 over Finite Field of size 3
sage: e = prod(G.gens())
sage: e
[2 0 0 0 0 1]
[2 0 0 0 0 0]
[0 2 0 0 0 0]
[0 0 2 0 0 0]
[0 0 0 2 0 0]
[0 0 0 0 2 0]
sage: v = V.an_element()
sage: v
2*e[0] + 2*e[1]
sage: e * v
e[0] + e[1] + e[2]
```

```
>>> from sage.all import *
>>> G = groups.matrix.SL(Integer(6), Integer(3))
>>> V = G.natural_representation()
>>> V
Natural representation of Special Linear Group of degree 6
 over Finite Field of size 3
>>> e = prod(G.gens())
>>> e
[2 0 0 0 0 1]
[2 0 0 0 0 0]
[0 2 0 0 0 0]
[0 0 2 0 0 0]
[0 0 0 2 0 0]
[0 0 0 0 2 0]
>>> v = V.an_element()
>>> v
2*e[0] + 2*e[1]
>>> e * v
```

(continues on next page)

(continued from previous page)

```
e[0] + e[1] + e[2]
```

sign_representation(*base_ring=None*)

Return the sign representation of `self` over `base_ring`.

INPUT:

- `base_ring` – (optional) the base ring; the default is the base ring of `self`

EXAMPLES:

```
sage: G = GL(2, QQ)
sage: V = G.sign_representation()
sage: e = G.an_element()
sage: e
[1 0]
[0 1]
sage: m2 = V.an_element()
sage: m2
2*B['v']
sage: m2*e
2*B['v']
sage: m2*e*e
2*B['v']

sage: W = WeylGroup(["A", 1, 1])
sage: W.sign_representation()
Sign representation of
 Weyl Group of type ['A', 1, 1] (as a matrix group acting on the root space)
 over Rational Field

sage: G = GL(4, 2)
sage: G.sign_representation() == G.trivial_representation()
True
```

```
>>> from sage.all import *
>>> G = GL(Integer(2), QQ)
>>> V = G.sign_representation()
>>> e = G.an_element()
>>> e
[1 0]
[0 1]
>>> m2 = V.an_element()
>>> m2
2*B['v']
>>> m2*e
2*B['v']
>>> m2*e*e
2*B['v']

>>> W = WeylGroup(["A", Integer(1), Integer(1)])
>>> W.sign_representation()
Sign representation of
```

(continues on next page)

(continued from previous page)

```
 Weyl Group of type ['A', 1, 1] (as a matrix group acting on the root space)
 over Rational Field

>>> G = GL(Integer(4), Integer(2))
>>> G.sign_representation() == G.trivial_representation()
True
```

subgroup(*generators*, *check=True*)

Return the subgroup generated by the given generators.

INPUT:

- `generators` – list/tuple/iterable of group elements of `self`
- `check` – boolean (default: `True`); whether to check that each matrix is invertible

OUTPUT: the subgroup generated by `generators` as an instance of `FinitelyGeneratedMatrixGroup_gap`

EXAMPLES:

```
sage: # needs sage.libs.gap sage.rings.number_field
sage: UCF = UniversalCyclotomicField()
sage: G  = GL(3, UCF)
sage: e3 = UCF.gen(3); e5 = UCF.gen(5)
sage: m = matrix(UCF, 3,3, [[e3, 1, 0], [0, e5, 7],[4, 3, 2]])
sage: S = G.subgroup([m]); S
Subgroup with 1 generators (
[E(3)    1    0]
[   0 E(5)    7]
[   4    3    2]
) of General Linear Group of degree 3 over Universal Cyclotomic Field

sage: # needs sage.rings.number_field
sage: CF3 = CyclotomicField(3)
sage: G  = GL(3, CF3)
sage: e3 = CF3.gen()
sage: m = matrix(CF3, 3,3, [[e3, 1, 0], [0, ~e3, 7],[4, 3, 2]])
sage: S = G.subgroup([m]); S
Subgroup with 1 generators (
[     zeta3           1           0]
[         0 -zeta3 - 1           7]
[         4           3           2]
) of General Linear Group of degree 3 over Cyclotomic Field of order 3 and␣
↪degree 2
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap sage.rings.number_field
>>> UCF = UniversalCyclotomicField()
>>> G  = GL(Integer(3), UCF)
>>> e3 = UCF.gen(Integer(3)); e5 = UCF.gen(Integer(5))
>>> m = matrix(UCF, Integer(3),Integer(3), [[e3, Integer(1), Integer(0)],␣
↪[Integer(0), e5, Integer(7)],[Integer(4), Integer(3), Integer(2)]])
>>> S = G.subgroup([m]); S
```

(continues on next page)

(continued from previous page)

```
Subgroup with 1 generators (
[E(3)    1    0]
[   0 E(5)    7]
[   4    3    2]
) of General Linear Group of degree 3 over Universal Cyclotomic Field

>>> # needs sage.rings.number_field
>>> CF3 = CyclotomicField(Integer(3))
>>> G  = GL(Integer(3), CF3)
>>> e3 = CF3.gen()
>>> m = matrix(CF3, Integer(3),Integer(3), [[e3, Integer(1), Integer(0)],
↪[Integer(0), ~e3, Integer(7)],[Integer(4), Integer(3), Integer(2)]])
>>> S = G.subgroup([m]); S
Subgroup with 1 generators (
[     zeta3          1          0]
[         0 -zeta3 - 1          7]
[         4          3          2]
) of General Linear Group of degree 3 over Cyclotomic Field of order 3 and
↪degree 2
```

class sage.groups.matrix_gps.matrix_group.**MatrixGroup_generic**(*degree*, *base_ring*, *category=None*)

Bases: *MatrixGroup_base*

Base class for matrix groups over generic base rings.

You should not use this class directly. Instead, use one of the more specialized derived classes.

INPUT:

- degree – integer; the degree (matrix size) of the matrix group
- base_ring – ring; the base ring of the matrices

Element

alias of *MatrixGroupElement_generic*

degree()

Return the degree of this matrix group.

OUTPUT: integer; the size (number of rows equals number of columns) of the matrices

EXAMPLES:

```
sage: SU(5,5).degree()                                                 #
↪needs sage.rings.finite_rings
5
```

```
>>> from sage.all import *
>>> SU(Integer(5),Integer(5)).degree()
↪                # needs sage.rings.finite_rings
5
```

is_trivial()

Return True if this group is the trivial group.

A group is trivial, if it consists only of the identity element, that is, if all its generators are the identity.

EXAMPLES:

```
sage: MatrixGroup([identity_matrix(3)]).is_trivial()
True
sage: SL(2, ZZ).is_trivial()
False
sage: CoxeterGroup(['B',3], implementation='matrix').is_trivial()
False
```

```
>>> from sage.all import *
>>> MatrixGroup([identity_matrix(Integer(3))]).is_trivial()
True
>>> SL(Integer(2), ZZ).is_trivial()
False
>>> CoxeterGroup(['B',Integer(3)], implementation='matrix').is_trivial()
False
```

matrix_space()

Return the matrix space corresponding to this matrix group.

This is a matrix space over the field of definition of this matrix group.

EXAMPLES:

```
sage: F = GF(5); MS = MatrixSpace(F, 2, 2)
sage: G = MatrixGroup([MS(1), MS([1,2,3,4])])
sage: G.matrix_space()
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 5
sage: G.matrix_space() is MS
True
```

```
>>> from sage.all import *
>>> F = GF(Integer(5)); MS = MatrixSpace(F, Integer(2), Integer(2))
>>> G = MatrixGroup([MS(Integer(1)), MS([Integer(1),Integer(2),Integer(3),
↪Integer(4)])])
>>> G.matrix_space()
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 5
>>> G.matrix_space() is MS
True
```

sage.groups.matrix_gps.matrix_group.**is_MatrixGroup**(*x*)

Test whether x is a matrix group.

EXAMPLES:

```
sage: from sage.groups.matrix_gps.matrix_group import is_MatrixGroup
sage: is_MatrixGroup(MatrixSpace(QQ, 3))
doctest:warning...
DeprecationWarning: the function is_MatrixGroup is deprecated;
use 'isinstance(..., MatrixGroup_base)' instead
See https://github.com/sagemath/sage/issues/37898 for details.
False
sage: is_MatrixGroup(Mat(QQ, 3))
False
```

(continues on next page)

(continued from previous page)

```
sage: is_MatrixGroup(GL(2, ZZ))
True
sage: is_MatrixGroup(MatrixGroup([matrix(2, [1,1,0,1])]))
True
```

```
>>> from sage.all import *
>>> from sage.groups.matrix_gps.matrix_group import is_MatrixGroup
>>> is_MatrixGroup(MatrixSpace(QQ, Integer(3)))
doctest:warning...
DeprecationWarning: the function is_MatrixGroup is deprecated;
use 'isinstance(..., MatrixGroup_base)' instead
See https://github.com/sagemath/sage/issues/37898 for details.
False
>>> is_MatrixGroup(Mat(QQ, Integer(3)))
False
>>> is_MatrixGroup(GL(Integer(2), ZZ))
True
>>> is_MatrixGroup(MatrixGroup([matrix(Integer(2), [Integer(1),Integer(1),
↪Integer(0),Integer(1)])]))
True
```

28.3 Matrix group over a ring that GAP understands

class sage.groups.matrix_gps.matrix_group_gap.**MatrixGroup_gap**(*degree*, *base_ring*, *libgap_group*, *ambient=None*, *category=None*)

Bases: *GroupMixinLibGAP*, *MatrixGroup_generic*, *ParentLibGAP*

Base class for matrix groups that implements GAP interface.

INPUT:

- degree – integer; the degree (matrix size) of the matrix group
- base_ring – ring; the base ring of the matrices
- libgap_group – the defining libgap group
- ambient – a derived class of *ParentLibGAP* or None (default); the ambient class if libgap_group has been defined as a subgroup

Element

alias of *MatrixGroupElement_gap*

structure_description(*G*, *latex=False*)

Return a string that tries to describe the structure of G.

This methods wraps GAP's StructureDescription method.

For full details, including the form of the returned string and the algorithm to build it, see GAP's documentation.

INPUT:

- latex – boolean (default: False); if True, return a LaTeX formatted string

OUTPUT: string

Warning

From GAP's documentation: The string returned by `StructureDescription` is **not** an isomorphism invariant: non-isomorphic groups can have the same string value, and two isomorphic groups in different representations can produce different strings.

EXAMPLES:

```
sage: # needs sage.groups
sage: G = CyclicPermutationGroup(6)
sage: G.structure_description()
'C6'
sage: G.structure_description(latex=True)
'C_{6}'
sage: G2 = G.direct_product(G, maps=False)
sage: LatexExpr(G2.structure_description(latex=True))
C_{6} \times C_{6}
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> G = CyclicPermutationGroup(Integer(6))
>>> G.structure_description()
'C6'
>>> G.structure_description(latex=True)
'C_{6}'
>>> G2 = G.direct_product(G, maps=False)
>>> LatexExpr(G2.structure_description(latex=True))
C_{6} \times C_{6}
```

This method is mainly intended for small groups or groups with few normal subgroups. Even then there are some surprises:

```
sage: D3 = DihedralGroup(3)
↪   # needs sage.groups
sage: D3.structure_description()
↪   # needs sage.groups
'S3'
```

```
>>> from sage.all import *
>>> D3 = DihedralGroup(Integer(3))
↪           # needs sage.groups
>>> D3.structure_description()
↪# needs sage.groups
'S3'
```

We use the Sage notation for the degree of dihedral groups:

```
sage: D4 = DihedralGroup(4)
↪   # needs sage.groups
sage: D4.structure_description()
↪   # needs sage.groups
'D4'
```

```
>>> from sage.all import *
>>> D4 = DihedralGroup(Integer(4))                                            ␣
↪         # needs sage.groups
>>> D4.structure_description()
↪# needs sage.groups
'D4'
```

Works for finitely presented groups (Issue #17573):

```
sage: F.<x, y> = FreeGroup()                                                   ␣
↪  # needs sage.groups
sage: G = F / [x^2*y^-1, x^3*y^2, x*y*x^-1*y^-1]                               ␣
↪  # needs sage.groups
sage: G.structure_description()                                                ␣
↪  # needs sage.groups
'C7'
```

```
>>> from sage.all import *
>>> F = FreeGroup(names=('x', 'y',)); (x, y,) = F._first_ngens(2)# needs sage.
↪groups
>>> G = F / [x**Integer(2)*y**-Integer(1), x**Integer(3)*y**Integer(2),␣
↪x*y*x**-Integer(1)*y**-Integer(1)]                               # needs␣
↪sage.groups
>>> G.structure_description()
↪# needs sage.groups
'C7'
```

And matrix groups (Issue #17573):

```
sage: groups.matrix.GL(4,2).structure_description()                           ␣
↪  # needs sage.libs.gap sage.modules
'A8'
```

```
>>> from sage.all import *
>>> groups.matrix.GL(Integer(4),Integer(2)).structure_description()           ␣
↪                    # needs sage.libs.gap sage.modules
'A8'
```

subgroup(*generators*, *check=True*)

Return the subgroup generated by the given generators.

INPUT:

- `generators` – list/tuple/iterable of group elements of `self`
- `check` – boolean (default: `True`); whether to check that each matrix is invertible

OUTPUT: the subgroup generated by `generators` as an instance of `FinitelyGeneratedMatrixGroup_gap`

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: UCF = UniversalCyclotomicField()
sage: G  = GL(3, UCF)
```

(continues on next page)

(continued from previous page)

```
sage: e3 = UCF.gen(3); e5 = UCF.gen(5)
sage: m = matrix(UCF, 3,3, [[e3, 1, 0], [0, e5, 7],[4, 3, 2]])
sage: S = G.subgroup([m]); S
Subgroup with 1 generators (
[E(3)    1    0]
[   0 E(5)    7]
[   4    3    2]
) of General Linear Group of degree 3 over Universal Cyclotomic Field

sage: # needs sage.rings.number_field
sage: CF3 = CyclotomicField(3)
sage: G  = GL(3, CF3)
sage: e3 = CF3.gen()
sage: m = matrix(CF3, 3,3, [[e3, 1, 0], [0, ~e3, 7],[4, 3, 2]])
sage: S = G.subgroup([m]); S
Subgroup with 1 generators (
[     zeta3          1          0]
[         0 -zeta3 - 1          7]
[         4          3          2]
) of General Linear Group of degree 3 over Cyclotomic Field of order 3 and␣
↪degree 2
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> UCF = UniversalCyclotomicField()
>>> G  = GL(Integer(3), UCF)
>>> e3 = UCF.gen(Integer(3)); e5 = UCF.gen(Integer(5))
>>> m = matrix(UCF, Integer(3),Integer(3), [[e3, Integer(1), Integer(0)],␣
↪[Integer(0), e5, Integer(7)],[Integer(4), Integer(3), Integer(2)]])
>>> S = G.subgroup([m]); S
Subgroup with 1 generators (
[E(3)    1    0]
[   0 E(5)    7]
[   4    3    2]
) of General Linear Group of degree 3 over Universal Cyclotomic Field

>>> # needs sage.rings.number_field
>>> CF3 = CyclotomicField(Integer(3))
>>> G  = GL(Integer(3), CF3)
>>> e3 = CF3.gen()
>>> m = matrix(CF3, Integer(3),Integer(3), [[e3, Integer(1), Integer(0)],␣
↪[Integer(0), ~e3, Integer(7)],[Integer(4), Integer(3), Integer(2)]])
>>> S = G.subgroup([m]); S
Subgroup with 1 generators (
[     zeta3          1          0]
[         0 -zeta3 - 1          7]
[         4          3          2]
) of General Linear Group of degree 3 over Cyclotomic Field of order 3 and␣
↪degree 2
```

28.4 Matrix Group Elements

EXAMPLES:

```
sage: F = GF(3); MS = MatrixSpace(F, 2, 2)
sage: gens = [MS([[1,0], [0,1]]), MS([[1,1], [0,1]])]
sage: G = MatrixGroup(gens); G
Matrix group over Finite Field of size 3 with 2 generators (
[1 0]  [1 1]
[0 1], [0 1] )
sage: g = G([[1,1], [0,1]])
sage: h = G([[1,2], [0,1]])
sage: g*h
[1 0]
[0 1]
```

```
>>> from sage.all import *
>>> F = GF(Integer(3)); MS = MatrixSpace(F, Integer(2), Integer(2))
>>> gens = [MS([[Integer(1),Integer(0)], [Integer(0),Integer(1)]]), MS([[Integer(1),
→Integer(1)], [Integer(0),Integer(1)]])]
>>> G = MatrixGroup(gens); G
Matrix group over Finite Field of size 3 with 2 generators (
[1 0]  [1 1]
[0 1], [0 1] )
>>> g = G([[Integer(1),Integer(1)], [Integer(0),Integer(1)]])
>>> h = G([[Integer(1),Integer(2)], [Integer(0),Integer(1)]])
>>> g*h
[1 0]
[0 1]
```

You cannot add two matrices, since this is not a group operation. You can coerce matrices back to the matrix space and add them there:

```
sage: g + h
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'Matrix group over Finite Field of size 3 with 2 generators (
[1 0]  [1 1]
[0 1], [0 1]
)' and
'Matrix group over Finite Field of size 3 with 2 generators (
[1 0]  [1 1]
[0 1], [0 1]
)'

sage: g.matrix() + h.matrix()
[2 0]
[0 2]
```

```
>>> from sage.all import *
>>> g + h
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: unsupported operand parent(s) for +:
'Matrix group over Finite Field of size 3 with 2 generators (
[1 0]  [1 1]
[0 1], [0 1]
)' and
'Matrix group over Finite Field of size 3 with 2 generators (
[1 0]  [1 1]
[0 1], [0 1]
)'

>>> g.matrix() + h.matrix()
[2 0]
[0 2]
```

Similarly, you cannot multiply group elements by scalars but you can do it with the underlying matrices:

```
sage: 2*g
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Integer Ring'
and 'Matrix group over Finite Field of size 3 with 2 generators (
[1 0]  [1 1]
[0 1], [0 1] )'
```

```
>>> from sage.all import *
>>> Integer(2)*g
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Integer Ring'
and 'Matrix group over Finite Field of size 3 with 2 generators (
[1 0]  [1 1]
[0 1], [0 1] )'
```

AUTHORS:

- David Joyner (2006-05): initial version David Joyner
- David Joyner (2006-05): various modifications to address William Stein's TODO's.
- William Stein (2006-12-09): many revisions.
- Volker Braun (2013-1) port to new Parent, libGAP.
- Travis Scrimshaw (2016-01): reworks class hierarchy in order to cythonize

class `sage.groups.matrix_gps.group_element.`**`MatrixGroupElement_generic`**

Bases: `MultiplicativeGroupElement`

Element of a matrix group over a generic ring.

The group elements are implemented as Sage matrices.

INPUT:

- `M` – a matrix
- `parent` – the parent

- `check` – boolean (default: `True`); if `True`, then do some type checking
- `convert` – boolean (default: `True`); if `True`, then convert `M` to the right matrix space

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], base_ring=ZZ)                              #␣
↪needs sage.graphs
sage: g = W.an_element(); g                                                 #␣
↪needs sage.graphs
[ 0  0 -1]
[ 1  0 -1]
[ 0  1 -1]
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(3)], base_ring=ZZ)                        ␣
↪     # needs sage.graphs
>>> g = W.an_element(); g                                                 #␣
↪needs sage.graphs
[ 0  0 -1]
[ 1  0 -1]
[ 0  1 -1]
```

inverse()

Return the inverse group element.

OUTPUT: a matrix group element

EXAMPLES:

```
sage: # needs sage.combinat sage.libs.gap
sage: W = CoxeterGroup(['A',3], base_ring=ZZ)
sage: g = W.an_element()
sage: ~g
[-1  1  0]
[-1  0  1]
[-1  0  0]
sage: g * ~g == W.one()
True
sage: ~g * g == W.one()
True

sage: # needs sage.combinat sage.libs.gap sage.rings.number_field
sage: W = CoxeterGroup(['B',3])
sage: W.base_ring()
Number Field in a with defining polynomial x^2 - 2 with a = 1.414213562373095?
sage: g = W.an_element()
sage: ~g
[-1  1  0]
[-1  0  a]
[-a  0  1]
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.libs.gap
>>> W = CoxeterGroup(['A',Integer(3)], base_ring=ZZ)
```

(continues on next page)

(continued from previous page)

```
>>> g = W.an_element()
>>> ~g
[-1  1  0]
[-1  0  1]
[-1  0  0]
>>> g * ~g == W.one()
True
>>> ~g * g == W.one()
True

>>> # needs sage.combinat sage.libs.gap sage.rings.number_field
>>> W = CoxeterGroup(['B',Integer(3)])
>>> W.base_ring()
Number Field in a with defining polynomial x^2 - 2 with a = 1.414213562373095?
>>> g = W.an_element()
>>> ~g
[-1  1  0]
[-1  0  a]
[-a  0  1]
```

is_one()

Return whether `self` is the identity of the group.

EXAMPLES:

```
sage: # needs sage.graphs
sage: W = CoxeterGroup(['A',3])
sage: g = W.gen(0)
sage: g.is_one()
False
sage: W.an_element().is_one()
False
sage: W.one().is_one()
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> W = CoxeterGroup(['A',Integer(3)])
>>> g = W.gen(Integer(0))
>>> g.is_one()
False
>>> W.an_element().is_one()
False
>>> W.one().is_one()
True
```

list()

Return list representation of this matrix.

EXAMPLES:

```
sage: # needs sage.combinat sage.libs.gap
sage: W = CoxeterGroup(['A',3], base_ring=ZZ)
```

(continues on next page)

(continued from previous page)

```
sage: g = W.gen(0)
sage: g
[-1  1  0]
[ 0  1  0]
[ 0  0  1]
sage: g.list()
[[-1, 1, 0], [0, 1, 0], [0, 0, 1]]
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.libs.gap
>>> W = CoxeterGroup(['A',Integer(3)], base_ring=ZZ)
>>> g = W.gen(Integer(0))
>>> g
[-1  1  0]
[ 0  1  0]
[ 0  0  1]
>>> g.list()
[[-1, 1, 0], [0, 1, 0], [0, 0, 1]]
```

matrix()

Obtain the usual matrix (as an element of a matrix space) associated to this matrix group element.

One reason to compute the associated matrix is that matrices support a huge range of functionality.

EXAMPLES:

```
sage: # needs sage.combinat sage.libs.gap
sage: W = CoxeterGroup(['A',3], base_ring=ZZ)
sage: g = W.gen(0)
sage: g.matrix()
[-1  1  0]
[ 0  1  0]
[ 0  0  1]
sage: parent(g.matrix())
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.libs.gap
>>> W = CoxeterGroup(['A',Integer(3)], base_ring=ZZ)
>>> g = W.gen(Integer(0))
>>> g.matrix()
[-1  1  0]
[ 0  1  0]
[ 0  0  1]
>>> parent(g.matrix())
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
```

Matrices have extra functionality that matrix group elements do not have:

```
sage: g.matrix().charpoly('t')                                          #␣
↪needs sage.combinat sage.libs.gap
t^3 - t^2 - t + 1
```

```
>>> from sage.all import *
>>> g.matrix().charpoly('t')                                              #␣
↪needs sage.combinat sage.libs.gap
t^3 - t^2 - t + 1
```

sage.groups.matrix_gps.group_element.**is_MatrixGroupElement**(*x*)

Test whether x is a matrix group element.

INPUT:

- x – anything

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.groups.matrix_gps.group_element import is_MatrixGroupElement
sage: is_MatrixGroupElement('helloooo')
False

sage: G = GL(2,3)
sage: is_MatrixGroupElement(G.an_element())
True
```

```
>>> from sage.all import *
>>> from sage.groups.matrix_gps.group_element import is_MatrixGroupElement
>>> is_MatrixGroupElement('helloooo')
False

>>> G = GL(Integer(2),Integer(3))
>>> is_MatrixGroupElement(G.an_element())
True
```

28.5 Matrix group elements implemented in GAP

class sage.groups.matrix_gps.group_element_gap.**MatrixGroupElement_gap**

Bases: *ElementLibGAP*

Element of a matrix group over a generic ring.

The group elements are implemented as wrappers around libGAP matrices.

INPUT:

- M – a matrix
- parent – the parent
- check – boolean (default: True); if True, do some type checking
- convert – boolean (default: True); if True, convert M to the right matrix space

list()

Return list representation of this matrix.

EXAMPLES:

```
sage: F = GF(3); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,0],[0,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: g = G.0
sage: g
[1 0]
[0 1]
sage: g.list()
[[1, 0], [0, 1]]
```

```
>>> from sage.all import *
>>> F = GF(Integer(3)); MS = MatrixSpace(F,Integer(2),Integer(2))
>>> gens = [MS([[Integer(1),Integer(0)],[Integer(0),Integer(1)]]),
↪MS([[Integer(1),Integer(1)],[Integer(0),Integer(1)]])]
>>> G = MatrixGroup(gens)
>>> g = G.gen(0)
>>> g
[1 0]
[0 1]
>>> g.list()
[[1, 0], [0, 1]]
```

matrix()

Obtain the usual matrix (as an element of a matrix space) associated to this matrix group element.

EXAMPLES:

```
sage: F = GF(3); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,0],[0,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: m = G.gen(0).matrix(); m
[1 0]
[0 1]
sage: m.parent()
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 3

sage: k = GF(7); G = MatrixGroup([matrix(k,2,[1,1,0,1]), matrix(k,2,[1,0,0,
↪2])])
sage: g = G.0
sage: g.matrix()
[1 1]
[0 1]
sage: parent(g.matrix())
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 7
```

```
>>> from sage.all import *
>>> F = GF(Integer(3)); MS = MatrixSpace(F,Integer(2),Integer(2))
>>> gens = [MS([[Integer(1),Integer(0)],[Integer(0),Integer(1)]]),
↪MS([[Integer(1),Integer(1)],[Integer(0),Integer(1)]])]
>>> G = MatrixGroup(gens)
>>> m = G.gen(Integer(0)).matrix(); m
[1 0]
[0 1]
```

(continues on next page)

(continued from previous page)

```
>>> m.parent()
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 3

>>> k = GF(Integer(7)); G = MatrixGroup([matrix(k,Integer(2),[Integer(1),
↪Integer(1),Integer(0),Integer(1)]), matrix(k,Integer(2),[Integer(1),
↪Integer(0),Integer(0),Integer(2)])])
>>> g = G.gen(0)
>>> g.matrix()
[1 1]
[0 1]
>>> parent(g.matrix())
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 7
```

Matrices have extra functionality that matrix group elements do not have:

```
sage: g.matrix().charpoly('t')
t^2 + 5*t + 1
```

```
>>> from sage.all import *
>>> g.matrix().charpoly('t')
t^2 + 5*t + 1
```

multiplicative_order()

Return the order of this group element, which is the smallest positive integer n such that $g^n = 1$, or +Infinity if no such integer exists.

EXAMPLES:

```
sage: k = GF(7)
sage: G = MatrixGroup([matrix(k,2,[1,1,0,1]), matrix(k,2,[1,0,0,2])]); G
Matrix group over Finite Field of size 7 with 2 generators (
[1 1]  [1 0]
[0 1], [0 2]
)
sage: G.order()
21
sage: G.gen(0).multiplicative_order(), G.gen(1).multiplicative_order()
(7, 3)
```

```
>>> from sage.all import *
>>> k = GF(Integer(7))
>>> G = MatrixGroup([matrix(k,Integer(2),[Integer(1),Integer(1),Integer(0),
↪Integer(1)]), matrix(k,Integer(2),[Integer(1),Integer(0),Integer(0),
↪Integer(2)])]); G
Matrix group over Finite Field of size 7 with 2 generators (
[1 1]  [1 0]
[0 1], [0 2]
)
>>> G.order()
21
>>> G.gen(Integer(0)).multiplicative_order(), G.gen(Integer(1)).
↪multiplicative_order()
```

(continues on next page)

(continued from previous page)

```
(7, 3)
```

`order` is just an alias for `multiplicative_order`:

```
sage: G.gen(0).order(), G.gen(1).order()
(7, 3)

sage: k = QQ
sage: G = MatrixGroup([matrix(k,2,[1,1,0,1]), matrix(k,2,[1,0,0,2])]); G
Matrix group over Rational Field with 2 generators (
[1 1]  [1 0]
[0 1], [0 2]
)
sage: G.order()
+Infinity
sage: G.gen(0).order(), G.gen(1).order()
(+Infinity, +Infinity)

sage: gl = GL(2, ZZ);  gl
General Linear Group of degree 2 over Integer Ring
sage: g = gl.gen(2);  g
[1 1]
[0 1]
sage: g.order()
+Infinity
```

```
>>> from sage.all import *
>>> G.gen(Integer(0)).order(), G.gen(Integer(1)).order()
(7, 3)

>>> k = QQ
>>> G = MatrixGroup([matrix(k,Integer(2),[Integer(1),Integer(1),Integer(0),
↪Integer(1)]), matrix(k,Integer(2),[Integer(1),Integer(0),Integer(0),
↪Integer(2)])]); G
Matrix group over Rational Field with 2 generators (
[1 1]  [1 0]
[0 1], [0 2]
)
>>> G.order()
+Infinity
>>> G.gen(Integer(0)).order(), G.gen(Integer(1)).order()
(+Infinity, +Infinity)

>>> gl = GL(Integer(2), ZZ);  gl
General Linear Group of degree 2 over Integer Ring
>>> g = gl.gen(Integer(2));  g
[1 1]
[0 1]
>>> g.order()
+Infinity
```

word_problem(*gens=None*)

Solve the word problem.

This method writes the group element as a product of the elements of the list `gens`, or the standard generators of the parent of `self` if `gens` is `None`.

INPUT:

- `gens` – list/tuple/iterable of elements (or objects that can be converted to group elements), or `None` (default); by default, the generators of the parent group are used

OUTPUT:

A factorization object that contains information about the order of factors and the exponents. A `ValueError` is raised if the group element cannot be written as a word in `gens`.

ALGORITHM:

Use GAP, which has optimized algorithms for solving the word problem (the GAP functions `EpimorphismFromFreeGroup` and `PreImagesRepresentative`).

EXAMPLES:

```
sage: G = GL(2,5); G
General Linear Group of degree 2 over Finite Field of size 5
sage: G.gens()
(
[2 0]  [4 1]
[0 1], [4 0]
)
sage: G(1).word_problem([G.gen(0)])
1
sage: type(_)
<class 'sage.structure.factorization.Factorization'>

sage: g = G([0,4,1,4])
sage: g.word_problem()
([4 1]
 [4 0])^-1
```

```
>>> from sage.all import *
>>> G = GL(Integer(2),Integer(5)); G
General Linear Group of degree 2 over Finite Field of size 5
>>> G.gens()
(
[2 0]  [4 1]
[0 1], [4 0]
)
>>> G(Integer(1)).word_problem([G.gen(Integer(0))])
1
>>> type(_)
<class 'sage.structure.factorization.Factorization'>

>>> g = G([Integer(0),Integer(4),Integer(1),Integer(4)])
>>> g.word_problem()
([4 1]
 [4 0])^-1
```

Next we construct a more complicated element of the group from the generators:

```
sage: s,t = G.0, G.1
sage: a = (s * t * s); b = a.word_problem(); b
([2 0]
 [0 1]) *
([4 1]
 [4 0]) *
([2 0]
 [0 1])
sage: flatten(b)
[
[2 0]     [4 1]     [2 0]
[0 1], 1, [4 0], 1, [0 1], 1
]
sage: b.prod() == a
True
```

```
>>> from sage.all import *
>>> s,t = G.gen(0), G.gen(1)
>>> a = (s * t * s); b = a.word_problem(); b
([2 0]
 [0 1]) *
([4 1]
 [4 0]) *
([2 0]
 [0 1])
>>> flatten(b)
[
[2 0]     [4 1]     [2 0]
[0 1], 1, [4 0], 1, [0 1], 1
]
>>> b.prod() == a
True
```

We solve the word problem using some different generators:

```
sage: s = G([2,0,0,1]); t = G([1,1,0,1]); u = G([0,-1,1,0])
sage: a.word_problem([s,t,u])
([2 0]
 [0 1])^-1 *
([1 1]
 [0 1])^-1 *
([0 4]
 [1 0]) *
([2 0]
 [0 1])^-1
```

```
>>> from sage.all import *
>>> s = G([Integer(2),Integer(0),Integer(0),Integer(1)]); t = G([Integer(1),
↪Integer(1),Integer(0),Integer(1)]); u = G([Integer(0),-Integer(1),
↪Integer(1),Integer(0)])
>>> a.word_problem([s,t,u])
([2 0]
```

(continues on next page)

(continued from previous page)

```
 [0 1])^-1 *
([1 1]
 [0 1])^-1 *
([0 4]
 [1 0]) *
([2 0]
 [0 1])^-1
```

We try some elements that don't actually generate the group:

```
sage: a.word_problem([t,u])
Traceback (most recent call last):
...
ValueError: word problem has no solution
```

```
>>> from sage.all import *
>>> a.word_problem([t,u])
Traceback (most recent call last):
...
ValueError: word problem has no solution
```

AUTHORS:

- David Joyner and William Stein
- David Loeffler (2010): fixed some bugs
- Volker Braun (2013): LibGAP

28.6 Finitely Generated Matrix Groups

This class is designed for computing with matrix groups defined by a finite set of generating matrices.

EXAMPLES:

```
sage: F = GF(3)
sage: gens = [matrix(F, 2, [1,0, -1,1]), matrix(F, 2, [1,1,0,1])]
sage: G = MatrixGroup(gens)
sage: G.conjugacy_classes_representatives()                                      #␣
↪needs sage.libs.gap
(
[1 0]  [0 2]  [0 1]  [2 0]  [0 2]  [0 1]  [0 2]
[0 1], [1 1], [2 1], [0 2], [1 2], [2 2], [1 0]
)
```

```
>>> from sage.all import *
>>> F = GF(Integer(3))
>>> gens = [matrix(F, Integer(2), [Integer(1),Integer(0), -Integer(1),Integer(1)]),␣
↪matrix(F, Integer(2), [Integer(1),Integer(1),Integer(0),Integer(1)])]
>>> G = MatrixGroup(gens)
>>> G.conjugacy_classes_representatives()                                    #␣
↪needs sage.libs.gap
(
```

(continues on next page)

(continued from previous page)

```
[1 0]   [0 2]   [0 1]   [2 0]   [0 2]   [0 1]   [0 2]
[0 1], [1 1], [2 1], [0 2], [1 2], [2 2], [1 0]
)
```

The finitely generated matrix groups can also be constructed as subgroups of matrix groups:

```
sage: SL2Z = SL(2, ZZ)
sage: S, T = SL2Z.gens()
sage: SL2Z.subgroup([T^2])
Subgroup with 1 generators (
[1 2]
[0 1]
) of Special Linear Group of degree 2 over Integer Ring
```

```
>>> from sage.all import *
>>> SL2Z = SL(Integer(2), ZZ)
>>> S, T = SL2Z.gens()
>>> SL2Z.subgroup([T**Integer(2)])
Subgroup with 1 generators (
[1 2]
[0 1]
) of Special Linear Group of degree 2 over Integer Ring
```

AUTHORS:

- William Stein: initial version
- David Joyner (2006-03-15): degree, base_ring, _contains_, list, random, order methods; examples
- William Stein (2006-12): rewrite
- David Joyner (2007-12): Added invariant_generators (with Martin Albrecht and Simon King)
- David Joyner (2008-08): Added module_composition_factors (interface to GAP's MeatAxe implementation) and as_permutation_group (returns isomorphic PermutationGroup).
- Simon King (2010-05): Improve invariant_generators by using GAP for the construction of the Reynolds operator in Singular.
- Volker Braun (2013-1) port to new Parent, libGAP.
- Sebastian Oehms (2018-07): Added _permutation_group_element_ (Issue #25706)
- Sebastian Oehms (2019-01): Revision of Issue #25706 (Issue #26903 and Issue #27143).

class sage.groups.matrix_gps.finitely_generated.**FinitelyGeneratedMatrixGroup_generic**(*degree*, *base_ring*, *generator_matrices*, *category=None*)

Bases: `MatrixGroup_generic`

gen (*i*)

Return the i-th generator.

OUTPUT: the i-th generator of the group

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: H = GL(2, GF(3))
sage: h1, h2 = H([[1,0], [2,1]]), H([[1,1], [0,1]])
sage: G = H.subgroup([h1, h2])
sage: G.gen(0)
[1 0]
[2 1]
sage: G.gen(0).matrix() == h1.matrix()
True
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> H = GL(Integer(2), GF(Integer(3)))
>>> h1, h2 = H([[Integer(1),Integer(0)], [Integer(2),Integer(1)]]),␣
↪H([[Integer(1),Integer(1)], [Integer(0),Integer(1)]])
>>> G = H.subgroup([h1, h2])
>>> G.gen(Integer(0))
[1 0]
[2 1]
>>> G.gen(Integer(0)).matrix() == h1.matrix()
True
```

gens ()

Return the generators of the matrix group.

EXAMPLES:

```
sage: F = GF(3); MS = MatrixSpace(F, 2, 2)
sage: gens = [MS([[1,0], [0,1]]), MS([[1,1], [0,1]])]
sage: G = MatrixGroup(gens)
sage: gens[0] in G
True
sage: gens = G.gens()
sage: gens[0] in G
True
sage: gens = [MS([[1,0], [0,1]]), MS([[1,1], [0,1]])]

sage: F = GF(5); MS = MatrixSpace(F, 2, 2)
sage: G = MatrixGroup([MS(1), MS([1,2, 3,4])])
sage: G
Matrix group over Finite Field of size 5 with 2 generators (
[1 0]  [1 2]
[0 1], [3 4]
)
sage: G.gens()
```

(continues on next page)

(continued from previous page)

```
(
[1 0]  [1 2]
[0 1], [3 4]
)
```

```
>>> from sage.all import *
>>> F = GF(Integer(3)); MS = MatrixSpace(F, Integer(2), Integer(2))
>>> gens = [MS([[Integer(1),Integer(0)], [Integer(0),Integer(1)]]),
↪MS([[Integer(1),Integer(1)], [Integer(0),Integer(1)]])]
>>> G = MatrixGroup(gens)
>>> gens[Integer(0)] in G
True
>>> gens = G.gens()
>>> gens[Integer(0)] in G
True
>>> gens = [MS([[Integer(1),Integer(0)], [Integer(0),Integer(1)]]),
↪MS([[Integer(1),Integer(1)], [Integer(0),Integer(1)]])]

>>> F = GF(Integer(5)); MS = MatrixSpace(F, Integer(2), Integer(2))
>>> G = MatrixGroup([MS(Integer(1)), MS([Integer(1),Integer(2), Integer(3),
↪Integer(4)])])
>>> G
Matrix group over Finite Field of size 5 with 2 generators (
[1 0]  [1 2]
[0 1], [3 4]
)
>>> G.gens()
(
[1 0]  [1 2]
[0 1], [3 4]
)
```

ngens()

Return the number of generators.

OUTPUT: integer; the number of generators

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: H = GL(2, GF(3))
sage: h1, h2 = H([[1,0], [2,1]]), H([[1,1], [0,1]])
sage: G = H.subgroup([h1, h2])
sage: G.ngens()
2
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> H = GL(Integer(2), GF(Integer(3)))
>>> h1, h2 = H([[Integer(1),Integer(0)], [Integer(2),Integer(1)]]),
↪H([[Integer(1),Integer(1)], [Integer(0),Integer(1)]])
>>> G = H.subgroup([h1, h2])
```

(continues on next page)

(continued from previous page)

```
>>> G.ngens()
2
```

sage.groups.matrix_gps.finitely_generated.**MatrixGroup**(**gens*, ***kwds*)

Return the matrix group with given generators.

INPUT:

- `*gens` – matrices, or a single list/tuple/iterable of matrices, or a matrix group
- `check` – boolean keyword argument (default: `True`); whether to check that each matrix is invertible

EXAMPLES:

```
sage: F = GF(5)
sage: gens = [matrix(F, 2, [1,2, -1,1]), matrix(F,2, [1,1, 0,1])]
sage: G = MatrixGroup(gens); G
Matrix group over Finite Field of size 5 with 2 generators (
[1 2]  [1 1]
[4 1], [0 1]
)
```

```
>>> from sage.all import *
>>> F = GF(Integer(5))
>>> gens = [matrix(F, Integer(2), [Integer(1),Integer(2), -Integer(1),
↪Integer(1)]), matrix(F,Integer(2), [Integer(1),Integer(1), Integer(0),
↪Integer(1)])]
>>> G = MatrixGroup(gens); G
Matrix group over Finite Field of size 5 with 2 generators (
[1 2]  [1 1]
[4 1], [0 1]
)
```

In the second example, the generators are a matrix over **Z**, a matrix over a finite field, and the integer 2. Sage determines that they both canonically map to matrices over the finite field, so creates that matrix group there:

```
sage: gens = [matrix(2, [1,2, -1,1]), matrix(GF(7), 2, [1,1, 0,1]), 2]
sage: G = MatrixGroup(gens); G
Matrix group over Finite Field of size 7 with 3 generators (
[1 2]  [1 1]  [2 0]
[6 1], [0 1], [0 2]
)
```

```
>>> from sage.all import *
>>> gens = [matrix(Integer(2), [Integer(1),Integer(2), -Integer(1),Integer(1)]),␣
↪matrix(GF(Integer(7)), Integer(2), [Integer(1),Integer(1), Integer(0),
↪Integer(1)]), Integer(2)]
>>> G = MatrixGroup(gens); G
Matrix group over Finite Field of size 7 with 3 generators (
[1 2]  [1 1]  [2 0]
[6 1], [0 1], [0 2]
)
```

Each generator must be invertible:

```
sage: G = MatrixGroup([matrix(ZZ, 2, [1,2,3,4])])
Traceback (most recent call last):
...
ValueError: each generator must be an invertible matrix

sage: F = GF(5); MS = MatrixSpace(F, 2, 2)
sage: MatrixGroup([MS.0])
Traceback (most recent call last):
...
ValueError: each generator must be an invertible matrix
sage: MatrixGroup([MS.0], check=False)  # works formally but is mathematical␣
↪nonsense
Matrix group over Finite Field of size 5 with 1 generators (
[1 0]
[0 0]
)
```

```
>>> from sage.all import *
>>> G = MatrixGroup([matrix(ZZ, Integer(2), [Integer(1),Integer(2),Integer(3),
↪Integer(4)])])
Traceback (most recent call last):
...
ValueError: each generator must be an invertible matrix

>>> F = GF(Integer(5)); MS = MatrixSpace(F, Integer(2), Integer(2))
>>> MatrixGroup([MS.gen(0)])
Traceback (most recent call last):
...
ValueError: each generator must be an invertible matrix
>>> MatrixGroup([MS.gen(0)], check=False)  # works formally but is mathematical␣
↪nonsense
Matrix group over Finite Field of size 5 with 1 generators (
[1 0]
[0 0]
)
```

Some groups are not supported, or do not have much functionality implemented:

```
sage: G = SL(0, QQ)
Traceback (most recent call last):
...
ValueError: the degree must be at least 1

sage: SL2C = SL(2, CC);  SL2C
Special Linear Group of degree 2 over Complex Field with 53 bits of precision
sage: SL2C.gens()
Traceback (most recent call last):
...
AttributeError: 'LinearMatrixGroup_generic_with_category' object has no attribute
↪'gens'...
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> G = SL(Integer(0), QQ)
Traceback (most recent call last):
...
ValueError: the degree must be at least 1

>>> SL2C = SL(Integer(2), CC);  SL2C
Special Linear Group of degree 2 over Complex Field with 53 bits of precision
>>> SL2C.gens()
Traceback (most recent call last):
...
AttributeError: 'LinearMatrixGroup_generic_with_category' object has no attribute
↪'gens'...
```

sage.groups.matrix_gps.finitely_generated.**QuaternionMatrixGroupGF3**()

The quaternion group as a set of 2×2 matrices over $\mathbf{F}_3$.

OUTPUT:

A matrix group consisting of 2×2 matrices with elements from the finite field of order 3. The group is the quaternion group, the nonabelian group of order 8 that is not isomorphic to the group of symmetries of a square (the dihedral group D_4).

Note

This group is most easily available via `groups.matrix.QuaternionGF3()`.

EXAMPLES:

The generators are the matrix representations of the elements commonly called I and J, while K is the product of I and J.

```
sage: from sage.groups.matrix_gps.finitely_generated import␣
↪QuaternionMatrixGroupGF3

sage: # needs sage.libs.gap
sage: Q = QuaternionMatrixGroupGF3()
sage: Q.order()
8
sage: aye = Q.gens()[0]; aye
[1 1]
[1 2]
sage: jay = Q.gens()[1]; jay
[2 1]
[1 1]
sage: kay = aye*jay; kay
[0 2]
[1 0]
```

```
>>> from sage.all import *
>>> from sage.groups.matrix_gps.finitely_generated import QuaternionMatrixGroupGF3

>>> # needs sage.libs.gap
```

(continues on next page)

(continued from previous page)

```
>>> Q = QuaternionMatrixGroupGF3()
>>> Q.order()
8
>>> aye = Q.gens()[Integer(0)]; aye
[1 1]
[1 2]
>>> jay = Q.gens()[Integer(1)]; jay
[2 1]
[1 1]
>>> kay = aye*jay; kay
[0 2]
[1 0]
```

sage.groups.matrix_gps.finitely_generated.**normalize_square_matrices**(*matrices*)

Find a common space for all matrices.

OUTPUT: list of matrices, all elements of the same matrix space

EXAMPLES:

```
sage: from sage.groups.matrix_gps.finitely_generated import normalize_square_
↪matrices
sage: m1 = [[1,2], [3,4]]
sage: m2 = [2, 3, 4, 5]
sage: m3 = matrix(QQ, [[1/2,1/3], [1/4,1/5]])
sage: m4 = MatrixGroup(m3).gen(0)
sage: normalize_square_matrices([m1, m2, m3, m4])
[
[1 2]  [2 3]  [1/2 1/3]  [1/2 1/3]
[3 4], [4 5], [1/4 1/5], [1/4 1/5]
]
```

```
>>> from sage.all import *
>>> from sage.groups.matrix_gps.finitely_generated import normalize_square_
↪matrices
>>> m1 = [[Integer(1),Integer(2)], [Integer(3),Integer(4)]]
>>> m2 = [Integer(2), Integer(3), Integer(4), Integer(5)]
>>> m3 = matrix(QQ, [[Integer(1)/Integer(2),Integer(1)/Integer(3)], [Integer(1)/
↪Integer(4),Integer(1)/Integer(5)]])
>>> m4 = MatrixGroup(m3).gen(Integer(0))
>>> normalize_square_matrices([m1, m2, m3, m4])
[
[1 2]  [2 3]  [1/2 1/3]  [1/2 1/3]
[3 4], [4 5], [1/4 1/5], [1/4 1/5]
]
```

28.7 Finitely Generated Matrix Groups with GAP

class sage.groups.matrix_gps.finitely_generated_gap.**FinitelyGeneratedMatrixGroup_gap**(*degree*, *base_ring*, *libgap_group*, *ambient=None*, *category=None*)

Bases: *MatrixGroup_gap*

Matrix group generated by a finite number of matrices.

EXAMPLES:

```
sage: m1 = matrix(GF(11), [[1,2],[3,4]])
sage: m2 = matrix(GF(11), [[1,3],[10,0]])
sage: G = MatrixGroup(m1, m2);  G
Matrix group over Finite Field of size 11 with 2 generators (
[1 2]  [ 1  3]
[3 4], [10  0]
)
sage: type(G)
<class 'sage.groups.matrix_gps.finitely_generated_gap.
↪FinitelyGeneratedMatrixGroup_gap_with_category'>
sage: TestSuite(G).run()
```

```
>>> from sage.all import *
>>> m1 = matrix(GF(Integer(11)), [[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]])
>>> m2 = matrix(GF(Integer(11)), [[Integer(1),Integer(3)],[Integer(10),
↪Integer(0)]])
>>> G = MatrixGroup(m1, m2);  G
Matrix group over Finite Field of size 11 with 2 generators (
[1 2]  [ 1  3]
[3 4], [10  0]
)
>>> type(G)
<class 'sage.groups.matrix_gps.finitely_generated_gap.
↪FinitelyGeneratedMatrixGroup_gap_with_category'>
>>> TestSuite(G).run()
```

as_permutation_group(*algorithm=None*, *seed=None*)

Return a permutation group representation for the group.

In most cases occurring in practice, this is a permutation group of minimal degree (the degree being determined from orbits under the group action). When these orbits are hard to compute, the procedure can be time-consuming and the degree may not be minimal.

INPUT:

- `algorithm` – `None` or `'smaller'`; in the latter case, try harder to find a permutation representation of small degree

- `seed` – `None` or an integer specifying the seed to fix results depending on pseudo-random-numbers; here it makes sense to be used with respect to the `'smaller'` option, since GAP produces random output in that context

OUTPUT:

A permutation group isomorphic to `self`. The `algorithm='smaller'` option tries to return an isomorphic group of low degree, but is not guaranteed to find the smallest one and must not even differ from the one obtained without the option. In that case repeating the invocation may help (see the example below).

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2), 5, 5)
sage: A = MS([[0,0,0,0,1],[0,0,0,1,0],[0,0,1,0,0],[0,1,0,0,0],[1,0,0,0,0]])
sage: G = MatrixGroup([A])
sage: G.as_permutation_group().order()
2
```

```
>>> from sage.all import *
>>> MS = MatrixSpace(GF(Integer(2)), Integer(5), Integer(5))
>>> A = MS([[Integer(0),Integer(0),Integer(0),Integer(0),Integer(1)],
↪[Integer(0),Integer(0),Integer(0),Integer(1),Integer(0)],[Integer(0),
↪Integer(0),Integer(1),Integer(0),Integer(0)],[Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(0)],[Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(0)]])
>>> G = MatrixGroup([A])
>>> G.as_permutation_group().order()
2
```

A finite subgroup of $GL(12, \mathbf{Z})$ as a permutation group:

```
sage: imf = libgap.function_factory('ImfMatrixGroup')
sage: GG = imf( 12, 3 )
sage: G = MatrixGroup(GG.GeneratorsOfGroup())
sage: G.cardinality()
21499084800
sage: P = G.as_permutation_group()
sage: Psmaller = G.as_permutation_group(algorithm='smaller', seed=6)
sage: P.cardinality()
21499084800
sage: P.degree()
144
sage: Psmaller.cardinality()
21499084800
sage: Psmaller.degree() <= P.degree()
True
```

```
>>> from sage.all import *
>>> imf = libgap.function_factory('ImfMatrixGroup')
>>> GG = imf( Integer(12), Integer(3) )
>>> G = MatrixGroup(GG.GeneratorsOfGroup())
>>> G.cardinality()
21499084800
>>> P = G.as_permutation_group()
>>> Psmaller = G.as_permutation_group(algorithm='smaller', seed=Integer(6))
```

(continues on next page)

(continued from previous page)

```
>>> P.cardinality()
21499084800
>>> P.degree()
144
>>> Psmaller.cardinality()
21499084800
>>> Psmaller.degree() <= P.degree()
True
```

Note

In this case, the "smaller" option returned an isomorphic group of lower degree. The above example used GAP's library of irreducible maximal finite ("imf") integer matrix groups to construct the *MatrixGroup* G over $\mathbf{F}_7$. The section "Irreducible Maximal Finite Integral Matrix Groups" in the GAP reference manual has more details.

Note

Concerning the option `algorithm='smaller'` you should note the following from GAP documentation: "The methods used might involve the use of random elements and the permutation representation (or even the degree of the representation) is not guaranteed to be the same for different calls of `SmallerDegreePermutationRepresentation`."

To obtain a reproducible result the optional argument `seed` may be used as in the example above.

invariant_generators()

Return invariant ring generators.

Computes generators for the polynomial ring $F[x_1, \ldots, x_n]^G$, where G in $GL(n, F)$ is a finite matrix group.

In the "good characteristic" case the polynomials returned form a minimal generating set for the algebra of G-invariant polynomials. In the "bad" case, the polynomials returned are primary and secondary invariants, forming a not necessarily minimal generating set for the algebra of G-invariant polynomials.

ALGORITHM:

Wraps Singular's `invariant_algebra_reynolds` and `invariant_ring` in `finvar.lib`.

EXAMPLES:

```
sage: F = GF(7); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[0,1],[-1,0]]),MS([[1,1],[2,3]])]
sage: G = MatrixGroup(gens)
sage: G.invariant_generators()                                          #␣
↪needs sage.libs.singular
[x1^7*x2 - x1*x2^7,
 x1^12 - 2*x1^9*x2^3 - x1^6*x2^6 + 2*x1^3*x2^9 + x2^12,
 x1^18 + 2*x1^15*x2^3 + 3*x1^12*x2^6 + 3*x1^6*x2^12 - 2*x1^3*x2^15 + x2^18]

sage: q = 4; a = 2
sage: MS = MatrixSpace(QQ, 2, 2)
sage: gen1 = [[1/a, (q-1)/a], [1/a, -1/a]]
```

(continues on next page)

(continued from previous page)

```
sage: gen2 = [[1,0], [0,-1]]; gen3 = [[-1,0], [0,1]]
sage: G = MatrixGroup([MS(gen1), MS(gen2), MS(gen3)])
sage: G.cardinality()
12
sage: G.invariant_generators()                                            #␣
↪needs sage.libs.singular
[x1^2 + 3*x2^2, x1^6 + 15*x1^4*x2^2 + 15*x1^2*x2^4 + 33*x2^6]

sage: # needs sage.rings.number_field
sage: F = CyclotomicField(8)
sage: z = F.gen()
sage: a = z+1/z
sage: b = z^2
sage: MS = MatrixSpace(F,2,2)
sage: g1 = MS([[1/a, 1/a], [1/a, -1/a]])
sage: g2 = MS([[-b, 0], [0, b]])
sage: G = MatrixGroup([g1,g2])
sage: G.invariant_generators()                                            #␣
↪needs sage.libs.singular
[x1^4 + 2*x1^2*x2^2 + x2^4,
 x1^5*x2 - x1*x2^5,
 x1^8 + 28/9*x1^6*x2^2 + 70/9*x1^4*x2^4 + 28/9*x1^2*x2^6 + x2^8]
```

```
>>> from sage.all import *
>>> F = GF(Integer(7)); MS = MatrixSpace(F,Integer(2),Integer(2))
>>> gens = [MS([[Integer(0),Integer(1)],[-Integer(1),Integer(0)]]),
↪MS([[Integer(1),Integer(1)],[Integer(2),Integer(3)]])]
>>> G = MatrixGroup(gens)
>>> G.invariant_generators()                                            #␣
↪needs sage.libs.singular
[x1^7*x2 - x1*x2^7,
 x1^12 - 2*x1^9*x2^3 - x1^6*x2^6 + 2*x1^3*x2^9 + x2^12,
 x1^18 + 2*x1^15*x2^3 + 3*x1^12*x2^6 + 3*x1^6*x2^12 - 2*x1^3*x2^15 + x2^18]

>>> q = Integer(4); a = Integer(2)
>>> MS = MatrixSpace(QQ, Integer(2), Integer(2))
>>> gen1 = [[Integer(1)/a, (q-Integer(1))/a], [Integer(1)/a, -Integer(1)/a]]
>>> gen2 = [[Integer(1),Integer(0)], [Integer(0),-Integer(1)]]; gen3 = [[-
↪Integer(1),Integer(0)], [Integer(0),Integer(1)]]
>>> G = MatrixGroup([MS(gen1), MS(gen2), MS(gen3)])
>>> G.cardinality()
12
>>> G.invariant_generators()                                            #␣
↪needs sage.libs.singular
[x1^2 + 3*x2^2, x1^6 + 15*x1^4*x2^2 + 15*x1^2*x2^4 + 33*x2^6]

>>> # needs sage.rings.number_field
>>> F = CyclotomicField(Integer(8))
>>> z = F.gen()
>>> a = z+Integer(1)/z
>>> b = z**Integer(2)
>>> MS = MatrixSpace(F,Integer(2),Integer(2))
```

(continues on next page)

(continued from previous page)

```
>>> g1 = MS([[Integer(1)/a, Integer(1)/a], [Integer(1)/a, -Integer(1)/a]])
>>> g2 = MS([[-b, Integer(0)], [Integer(0), b]])
>>> G = MatrixGroup([g1,g2])
>>> G.invariant_generators()                                         #␣
↪needs sage.libs.singular
[x1^4 + 2*x1^2*x2^2 + x2^4,
 x1^5*x2 - x1*x2^5,
 x1^8 + 28/9*x1^6*x2^2 + 70/9*x1^4*x2^4 + 28/9*x1^2*x2^6 + x2^8]
```

AUTHORS:

- David Joyner, Simon King and Martin Albrecht.

REFERENCES:

- Singular reference manual
- [Stu1993]
- S. King, "Minimal Generating Sets of non-modular invariant rings of finite groups", arXiv math/0703035.

invariants_of_degree(*deg*, *chi=None*, *R=None*)

Return the (relative) invariants of given degree for this group.

For this group, compute the invariants of degree `deg` with respect to the group character `chi`. The method is to project each possible monomial of degree `deg` via the Reynolds operator. Note that if the polynomial ring `R` is specified it's base ring may be extended if the resulting invariant is defined over a bigger field.

INPUT:

- `degree` – positive integer
- `chi` – (default: trivial character) a linear group character of this group
- `R` – (optional) a polynomial ring

OUTPUT: list of polynomials

EXAMPLES:

```
sage: # needs sage.groups sage.rings.number_field
sage: Gr = MatrixGroup(SymmetricGroup(2))
sage: sorted(Gr.invariants_of_degree(3))
[x0^2*x1 + x0*x1^2, x0^3 + x1^3]
sage: R.<x,y> = QQ[]
sage: sorted(Gr.invariants_of_degree(4, R=R))
[x^2*y^2, x^3*y + x*y^3, x^4 + y^4]
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> Gr = MatrixGroup(SymmetricGroup(Integer(2)))
>>> sorted(Gr.invariants_of_degree(Integer(3)))
[x0^2*x1 + x0*x1^2, x0^3 + x1^3]
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> sorted(Gr.invariants_of_degree(Integer(4), R=R))
[x^2*y^2, x^3*y + x*y^3, x^4 + y^4]
```

```
sage: # needs sage.groups sage.rings.number_field
sage: R.<x,y,z> = QQ[]
sage: Gr = MatrixGroup(DihedralGroup(3))
sage: ct = Gr.character_table()
sage: chi = Gr.character(ct[0])
sage: all(f(*(g.matrix()*vector(R.gens()))) == chi(g)*f
....: for f in Gr.invariants_of_degree(3, R=R, chi=chi) for g in Gr)
True
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> Gr = MatrixGroup(DihedralGroup(Integer(3)))
>>> ct = Gr.character_table()
>>> chi = Gr.character(ct[Integer(0)])
>>> all(f(*(g.matrix()*vector(R.gens()))) == chi(g)*f
... for f in Gr.invariants_of_degree(Integer(3), R=R, chi=chi) for g in Gr)
True
```

```
sage: i = GF(7)(3)
sage: G = MatrixGroup([[i^3,0,0,-i^3],[i^2,0,0,-i^2]])
sage: G.invariants_of_degree(25)                                        #␣
↪needs sage.rings.number_field
[]
```

```
>>> from sage.all import *
>>> i = GF(Integer(7))(Integer(3))
>>> G = MatrixGroup([[i**Integer(3),Integer(0),Integer(0),-i**Integer(3)],
↪[i**Integer(2),Integer(0),Integer(0),-i**Integer(2)]])
>>> G.invariants_of_degree(Integer(25))                                 ␣
↪     # needs sage.rings.number_field
[]
```

```
sage: # needs sage.groups
sage: G = MatrixGroup(SymmetricGroup(5))
sage: R = QQ['x,y']
sage: G.invariants_of_degree(3, R=R)
Traceback (most recent call last):
...
TypeError: number of variables in polynomial ring must match size of matrices
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> G = MatrixGroup(SymmetricGroup(Integer(5)))
>>> R = QQ['x,y']
>>> G.invariants_of_degree(Integer(3), R=R)
Traceback (most recent call last):
...
TypeError: number of variables in polynomial ring must match size of matrices
```

```
sage: # needs sage.groups sage.rings.number_field
sage: K.<i> = CyclotomicField(4)
sage: G =  MatrixGroup(CyclicPermutationGroup(3))
sage: chi = G.character(G.character_table()[1])
sage: R.<x,y,z> = K[]
sage: sorted(G.invariants_of_degree(2, R=R, chi=chi))
[x*y + (-2*izeta3^3 - 3*izeta3^2 - 8*izeta3 - 4)*x*z
     + (2*izeta3^3 + 3*izeta3^2 + 8*izeta3 + 3)*y*z,
 x^2 + (2*izeta3^3 + 3*izeta3^2 + 8*izeta3 + 3)*y^2
     + (-2*izeta3^3 - 3*izeta3^2 - 8*izeta3 - 4)*z^2]
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> K = CyclotomicField(Integer(4), names=('i',)); (i,) = K._first_ngens(1)
>>> G =  MatrixGroup(CyclicPermutationGroup(Integer(3)))
>>> chi = G.character(G.character_table()[Integer(1)])
>>> R = K['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> sorted(G.invariants_of_degree(Integer(2), R=R, chi=chi))
[x*y + (-2*izeta3^3 - 3*izeta3^2 - 8*izeta3 - 4)*x*z
     + (2*izeta3^3 + 3*izeta3^2 + 8*izeta3 + 3)*y*z,
 x^2 + (2*izeta3^3 + 3*izeta3^2 + 8*izeta3 + 3)*y^2
     + (-2*izeta3^3 - 3*izeta3^2 - 8*izeta3 - 4)*z^2]
```

```
sage: # needs sage.groups sage.rings.number_field
sage: S3 = MatrixGroup(SymmetricGroup(3))
sage: chi = S3.character(S3.character_table()[0])
sage: sorted(S3.invariants_of_degree(5, chi=chi))
[x0^3*x1^2 - x0^2*x1^3 - x0^3*x2^2 + x1^3*x2^2 + x0^2*x2^3 - x1^2*x2^3,
 x0^4*x1 - x0*x1^4 - x0^4*x2 + x1^4*x2 + x0*x2^4 - x1*x2^4]
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> S3 = MatrixGroup(SymmetricGroup(Integer(3)))
>>> chi = S3.character(S3.character_table()[Integer(0)])
>>> sorted(S3.invariants_of_degree(Integer(5), chi=chi))
[x0^3*x1^2 - x0^2*x1^3 - x0^3*x2^2 + x1^3*x2^2 + x0^2*x2^3 - x1^2*x2^3,
 x0^4*x1 - x0*x1^4 - x0^4*x2 + x1^4*x2 + x0*x2^4 - x1*x2^4]
```

module_composition_factors(*algorithm=None*)

Return a list of triples consisting of [base field, dimension, irreducibility], for each of the Meataxe composition factors modules. The `algorithm="verbose"` option returns more information, but in Meataxe notation.

EXAMPLES:

```
sage: F = GF(3); MS = MatrixSpace(F,4,4)
sage: M = MS(0)
sage: M[0,1] = 1; M[1,2] = 1; M[2,3] = 1; M[3,0] = 1
sage: G = MatrixGroup([M])
sage: G.module_composition_factors()
[(Finite Field of size 3, 1, True),
 (Finite Field of size 3, 1, True),
 (Finite Field of size 3, 2, True)]
```

(continues on next page)

(continued from previous page)

```
sage: F = GF(7); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[0,1],[-1,0]]), MS([[1,1],[2,3]])]
sage: G = MatrixGroup(gens)
sage: G.module_composition_factors()
[(Finite Field of size 7, 2, True)]
```

```
>>> from sage.all import *
>>> F = GF(Integer(3)); MS = MatrixSpace(F,Integer(4),Integer(4))
>>> M = MS(Integer(0))
>>> M[Integer(0),Integer(1)] = Integer(1); M[Integer(1),Integer(2)] =␣
↪Integer(1); M[Integer(2),Integer(3)] = Integer(1); M[Integer(3),Integer(0)]␣
↪= Integer(1)
>>> G = MatrixGroup([M])
>>> G.module_composition_factors()
[(Finite Field of size 3, 1, True),
 (Finite Field of size 3, 1, True),
 (Finite Field of size 3, 2, True)]
>>> F = GF(Integer(7)); MS = MatrixSpace(F,Integer(2),Integer(2))
>>> gens = [MS([[Integer(0),Integer(1)],[-Integer(1),Integer(0)]]),␣
↪MS([[Integer(1),Integer(1)],[Integer(2),Integer(3)]])]
>>> G = MatrixGroup(gens)
>>> G.module_composition_factors()
[(Finite Field of size 7, 2, True)]
```

Type `G.module_composition_factors(algorithm='verbose')` to get a more verbose version.

For more on MeatAxe notation, see GAP: chap69.

molien_series(*chi=None*, *return_series=True*, *prec=20*, *variable='t'*)

Compute the Molien series of this finite group with respect to the character `chi`.

It can be returned either as a rational function in one variable or a power series in one variable. The base field must be a finite field, the rationals, or a cyclotomic field.

Note that the base field characteristic cannot divide the group order (i.e., the non-modular case).

ALGORITHM:

For a finite group G in characteristic zero we construct the Molien series as

$$\frac{1}{|G|}\sum_{g\in G}\frac{\chi(g)}{\det(I-tg)},$$

where I is the identity matrix and t an indeterminate.

For characteristic p not dividing the order of G, let k be the base field and N the order of G. Define λ as a primitive N-th root of unity over k and ω as a primitive N-th root of unity over $\mathbf{Q}$. For each $g \in G$ define $k_i(g)$ to be the positive integer such that $e_i = \lambda^{k_i(g)}$ for each eigenvalue e_i of g. Then the Molien series is computed as

$$\frac{1}{|G|}\sum_{g\in G}\frac{\chi(g)}{\prod_{i=1}^{n}(1-t\omega^{k_i(g)})},$$

where t is an indeterminant. [Dec1998]

INPUT:

- `chi` – (default: trivial character) a linear group character of this group
- `return_series` – boolean (default: `True`); if `True`, then returns the Molien series as a power series, `False` as a rational function
- `prec` – integer (default: 20); power series default precision (possibly infinite, in which case it is computed lazily)
- `variable` – string (default: `'t'`); variable name for the Molien series

OUTPUT: single variable rational function or power series with integer coefficients

EXAMPLES:

```
sage: MatrixGroup(matrix(QQ,2,2,[1,1,0,1])).molien_series()
Traceback (most recent call last):
...
NotImplementedError: only implemented for finite groups
sage: MatrixGroup(matrix(GF(3),2,2,[1,1,0,1])).molien_series()              #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
NotImplementedError: characteristic cannot divide group order
```

```
>>> from sage.all import *
>>> MatrixGroup(matrix(QQ,Integer(2),Integer(2),[Integer(1),Integer(1),
↪Integer(0),Integer(1)])).molien_series()
Traceback (most recent call last):
...
NotImplementedError: only implemented for finite groups
>>> MatrixGroup(matrix(GF(Integer(3)),Integer(2),Integer(2),[Integer(1),
↪Integer(1),Integer(0),Integer(1)])).molien_series()              # needs␣
↪sage.rings.number_field
Traceback (most recent call last):
...
NotImplementedError: characteristic cannot divide group order
```

Tetrahedral Group:

```
sage: # needs sage.rings.number_field
sage: K.<i> = CyclotomicField(4)
sage: Tetra =  MatrixGroup([(-1+i)/2,(-1+i)/2, (1+i)/2,(-1-i)/2], [0,i, -i,0])
sage: Tetra.molien_series(prec=30)
1 + t^8 + 2*t^12 + t^16 + 2*t^20 + 3*t^24 + 2*t^28 + O(t^30)
sage: mol = Tetra.molien_series(return_series=False); mol
(t^8 - t^4 + 1)/(t^16 - t^12 - t^4 + 1)
sage: mol.parent()
Fraction Field of Univariate Polynomial Ring in t over Integer Ring
sage: chi = Tetra.character(Tetra.character_table()[1])
sage: Tetra.molien_series(chi, prec=30, variable='u')
u^6 + u^14 + 2*u^18 + u^22 + 2*u^26 + 3*u^30 + 2*u^34 + O(u^36)
sage: chi = Tetra.character(Tetra.character_table()[2])
sage: Tetra.molien_series(chi)
t^10 + t^14 + t^18 + 2*t^22 + 2*t^26 + O(t^30)
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> K = CyclotomicField(Integer(4), names=('i',)); (i,) = K._first_ngens(1)
>>> Tetra =  MatrixGroup([(-Integer(1)+i)/Integer(2),(-Integer(1)+i)/
↪Integer(2), (Integer(1)+i)/Integer(2),(-Integer(1)-i)/Integer(2)],
↪[Integer(0),i, -i,Integer(0)])
>>> Tetra.molien_series(prec=Integer(30))
1 + t^8 + 2*t^12 + t^16 + 2*t^20 + 3*t^24 + 2*t^28 + O(t^30)
>>> mol = Tetra.molien_series(return_series=False); mol
(t^8 - t^4 + 1)/(t^16 - t^12 - t^4 + 1)
>>> mol.parent()
Fraction Field of Univariate Polynomial Ring in t over Integer Ring
>>> chi = Tetra.character(Tetra.character_table()[Integer(1)])
>>> Tetra.molien_series(chi, prec=Integer(30), variable='u')
u^6 + u^14 + 2*u^18 + u^22 + 2*u^26 + 3*u^30 + 2*u^34 + O(u^36)
>>> chi = Tetra.character(Tetra.character_table()[Integer(2)])
>>> Tetra.molien_series(chi)
t^10 + t^14 + t^18 + 2*t^22 + 2*t^26 + O(t^30)
```

```
sage: # needs sage.groups sage.rings.number_field
sage: S3 = MatrixGroup(SymmetricGroup(3))
sage: mol = S3.molien_series(prec=10); mol
1 + t + 2*t^2 + 3*t^3 + 4*t^4 + 5*t^5 + 7*t^6 + 8*t^7 + 10*t^8 + 12*t^9 + O(t^
↪10)
sage: mol.parent()
Power Series Ring in t over Integer Ring
sage: mol = S3.molien_series(prec=oo); mol
1 + t + 2*t^2 + 3*t^3 + 4*t^4 + 5*t^5 + 7*t^6 + O(t^7)
sage: mol.parent()
Lazy Taylor Series Ring in t over Integer Ring
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> S3 = MatrixGroup(SymmetricGroup(Integer(3)))
>>> mol = S3.molien_series(prec=Integer(10)); mol
1 + t + 2*t^2 + 3*t^3 + 4*t^4 + 5*t^5 + 7*t^6 + 8*t^7 + 10*t^8 + 12*t^9 + O(t^
↪10)
>>> mol.parent()
Power Series Ring in t over Integer Ring
>>> mol = S3.molien_series(prec=oo); mol
1 + t + 2*t^2 + 3*t^3 + 4*t^4 + 5*t^5 + 7*t^6 + O(t^7)
>>> mol.parent()
Lazy Taylor Series Ring in t over Integer Ring
```

Octahedral Group:

```
sage: # needs sage.rings.number_field
sage: K.<v> = CyclotomicField(8)
sage: a = v - v^3  # sqrt(2)
sage: i = v^2
sage: Octa = MatrixGroup([(-1+i)/2, (-1+i)/2,  (1+i)/2, (-1-i)/2],          #␣
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
....:                     [(1+i)/a, 0,  0, (1-i)/a])
sage: Octa.molien_series(prec=30)                                            #␣
↪needs sage.symbolic
1 + t^8 + t^12 + t^16 + t^18 + t^20 + 2*t^24 + t^26 + t^28 + O(t^30)
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> K = CyclotomicField(Integer(8), names=('v',)); (v,) = K._first_ngens(1)
>>> a = v - v**Integer(3)  # sqrt(2)
>>> i = v**Integer(2)
>>> Octa = MatrixGroup([(-Integer(1)+i)/Integer(2), (-Integer(1)+i)/
↪Integer(2),  (Integer(1)+i)/Integer(2), (-Integer(1)-i)/Integer(2)],          ␣
↪   # needs sage.symbolic
...                     [(Integer(1)+i)/a, Integer(0),  Integer(0),␣
↪(Integer(1)-i)/a])
>>> Octa.molien_series(prec=Integer(30))                                      ␣
↪      # needs sage.symbolic
1 + t^8 + t^12 + t^16 + t^18 + t^20 + 2*t^24 + t^26 + t^28 + O(t^30)
```

Icosahedral Group:

```
sage: # needs sage.rings.number_field
sage: K.<v> = CyclotomicField(10)
sage: z5 = v^2
sage: i = z5^5
sage: a = 2*z5^3 + 2*z5^2 + 1  #sqrt(5)
sage: Ico = MatrixGroup([[z5^3,0, 0,z5^2],
....:                     [0,1, -1,0],
....:                     [(z5^4-z5)/a, (z5^2-z5^3)/a,
....:                      (z5^2-z5^3)/a, -(z5^4-z5)/a]])
sage: Ico.molien_series(prec=40)
1 + t^12 + t^20 + t^24 + t^30 + t^32 + t^36 + O(t^40)
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> K = CyclotomicField(Integer(10), names=('v',)); (v,) = K._first_ngens(1)
>>> z5 = v**Integer(2)
>>> i = z5**Integer(5)
>>> a = Integer(2)*z5**Integer(3) + Integer(2)*z5**Integer(2) + Integer(1)
↪#sqrt(5)
>>> Ico = MatrixGroup([[z5**Integer(3),Integer(0), Integer(0),z5**Integer(2)],
...                     [Integer(0),Integer(1), -Integer(1),Integer(0)],
...                     [(z5**Integer(4)-z5)/a, (z5**Integer(2)-
↪z5**Integer(3))/a,
...                      (z5**Integer(2)-z5**Integer(3))/a, -(z5**Integer(4)-
↪z5)/a]])
>>> Ico.molien_series(prec=Integer(40))
1 + t^12 + t^20 + t^24 + t^30 + t^32 + t^36 + O(t^40)
```

```
sage: # needs sage.groups sage.rings.number_field
sage: G = MatrixGroup(CyclicPermutationGroup(3))
```

(continues on next page)

(continued from previous page)

```
sage: chi = G.character(G.character_table()[1])
sage: G.molien_series(chi, prec=10)
t + 2*t^2 + 3*t^3 + 5*t^4 + 7*t^5 + 9*t^6
 + 12*t^7 + 15*t^8 + 18*t^9 + 22*t^10 + O(t^11)
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> G = MatrixGroup(CyclicPermutationGroup(Integer(3)))
>>> chi = G.character(G.character_table()[Integer(1)])
>>> G.molien_series(chi, prec=Integer(10))
t + 2*t^2 + 3*t^3 + 5*t^4 + 7*t^5 + 9*t^6
 + 12*t^7 + 15*t^8 + 18*t^9 + 22*t^10 + O(t^11)
```

```
sage: # needs sage.groups sage.rings.number_field
sage: K = GF(5)
sage: S = MatrixGroup(SymmetricGroup(4))
sage: G = MatrixGroup([matrix(K, 4, 4,
....:                             [K(y) for u in m.list() for y in u])
....:                     for m in S.gens()])
sage: G.molien_series(return_series=False)
1/(t^10 - t^9 - t^8 + 2*t^5 - t^2 - t + 1)
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> K = GF(Integer(5))
>>> S = MatrixGroup(SymmetricGroup(Integer(4)))
>>> G = MatrixGroup([matrix(K, Integer(4), Integer(4),
...                             [K(y) for u in m.list() for y in u])
...                     for m in S.gens()])
>>> G.molien_series(return_series=False)
1/(t^10 - t^9 - t^8 + 2*t^5 - t^2 - t + 1)
```

```
sage: # needs sage.rings.number_field
sage: i = GF(7)(3)
sage: G = MatrixGroup([[i^3,0, 0,-i^3], [i^2,0, 0,-i^2]])
sage: chi = G.character(G.character_table()[4])
sage: G.molien_series(chi)
3*t^5 + 6*t^11 + 9*t^17 + 12*t^23 + O(t^25)
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> i = GF(Integer(7))(Integer(3))
>>> G = MatrixGroup([[i**Integer(3),Integer(0), Integer(0),-i**Integer(3)],
↪[i**Integer(2),Integer(0), Integer(0),-i**Integer(2)]])
>>> chi = G.character(G.character_table()[Integer(4)])
>>> G.molien_series(chi)
3*t^5 + 6*t^11 + 9*t^17 + 12*t^23 + O(t^25)
```

reynolds_operator(*poly*, *chi=None*)

Compute the Reynolds operator of this finite group G.

This is the projection from a polynomial ring to the ring of relative invariants [Stu1993]. If possible, the invariant is returned defined over the base field of the given polynomial `poly`, otherwise, it is returned over the compositum of the fields involved in the computation. Only implemented for absolute fields.

ALGORITHM:

Let $K[x]$ be a polynomial ring and χ a linear character for G. Let

be the ring of invariants of G relative to χ. Then the Reynolds operator is a map R from $K[x]$ into $K[x]^G_\chi$ defined by

INPUT:

- `poly` – a polynomial
- `chi` – (default: trivial character) a linear group character of this group

OUTPUT: an invariant polynomial relative to χ

AUTHORS:

Rebecca Lauren Miller and Ben Hutz

EXAMPLES:

```
sage: S3 = MatrixGroup(SymmetricGroup(3))
sage: R.<x,y,z> = QQ[]
sage: f = x*y*z^3
sage: S3.reynolds_operator(f)                                                #␣
↪needs sage.rings.number_field
1/3*x^3*y*z + 1/3*x*y^3*z + 1/3*x*y*z^3
```

```
>>> from sage.all import *
>>> S3 = MatrixGroup(SymmetricGroup(Integer(3)))
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> f = x*y*z**Integer(3)
>>> S3.reynolds_operator(f)                                                #␣
↪needs sage.rings.number_field
1/3*x^3*y*z + 1/3*x*y^3*z + 1/3*x*y*z^3
```

```
sage: # needs sage.groups sage.rings.number_field
sage: G = MatrixGroup(CyclicPermutationGroup(4))
sage: chi = G.character(G.character_table()[3])
sage: K.<v> = CyclotomicField(4)
sage: R.<x,y,z,w> = K[]
sage: G.reynolds_operator(x, chi)
1/4*x + (1/4*v)*y - 1/4*z + (-1/4*v)*w
sage: chi = G.character(G.character_table()[2])
sage: R.<x,y,z,w> = QQ[]
sage: G.reynolds_operator(x*y, chi)
1/4*x*y + (-1/4*zeta4)*y*z + (1/4*zeta4)*x*w - 1/4*z*w
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> G = MatrixGroup(CyclicPermutationGroup(Integer(4)))
>>> chi = G.character(G.character_table()[Integer(3)])
>>> K = CyclotomicField(Integer(4), names=('v',)); (v,) = K._first_ngens(1)
>>> R = K['x, y, z, w']; (x, y, z, w,) = R._first_ngens(4)
```

(continues on next page)

(continued from previous page)

```
>>> G.reynolds_operator(x, chi)
1/4*x + (1/4*v)*y - 1/4*z + (-1/4*v)*w
>>> chi = G.character(G.character_table()[Integer(2)])
>>> R = QQ['x, y, z, w']; (x, y, z, w,) = R._first_ngens(4)
>>> G.reynolds_operator(x*y, chi)
1/4*x*y + (-1/4*zeta4)*y*z + (1/4*zeta4)*x*w - 1/4*z*w
```

```
sage: # needs sage.groups sage.rings.number_field
sage: K.<i> = CyclotomicField(4)
sage: G =  MatrixGroup(CyclicPermutationGroup(3))
sage: chi = G.character(G.character_table()[1])
sage: R.<x,y,z> = K[]
sage: G.reynolds_operator(x*y^5, chi)
1/3*x*y^5 + (-2/3*izeta3^3 - izeta3^2 - 8/3*izeta3 - 4/3)*x^5*z
          + (2/3*izeta3^3 + izeta3^2 + 8/3*izeta3 + 1)*y*z^5
sage: R.<x,y,z> = QQbar[]
sage: G.reynolds_operator(x*y^5, chi)
 1/3*x*y^5 + (-0.1666666666666667? + 0.2886751345948129?*I)*x^5*z
           + (-0.1666666666666667? - 0.2886751345948129?*I)*y*z^5
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> K = CyclotomicField(Integer(4), names=('i',)); (i,) = K._first_ngens(1)
>>> G =  MatrixGroup(CyclicPermutationGroup(Integer(3)))
>>> chi = G.character(G.character_table()[Integer(1)])
>>> R = K['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> G.reynolds_operator(x*y**Integer(5), chi)
1/3*x*y^5 + (-2/3*izeta3^3 - izeta3^2 - 8/3*izeta3 - 4/3)*x^5*z
          + (2/3*izeta3^3 + izeta3^2 + 8/3*izeta3 + 1)*y*z^5
>>> R = QQbar['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> G.reynolds_operator(x*y**Integer(5), chi)
 1/3*x*y^5 + (-0.1666666666666667? + 0.2886751345948129?*I)*x^5*z
           + (-0.1666666666666667? - 0.2886751345948129?*I)*y*z^5
```

```
sage: # needs sage.rings.number_field
sage: K.<i> = CyclotomicField(4)
sage: Tetra =  MatrixGroup([(-1+i)/2,(-1+i)/2, (1+i)/2,(-1-i)/2], [0,i, -i,0])
sage: chi = Tetra.character(Tetra.character_table()[4])
sage: L.<v> = QuadraticField(-3)
sage: R.<x,y> = L[]
sage: Tetra.reynolds_operator(x^4)
0
sage: Tetra.reynolds_operator(x^4, chi)
1/4*x^4 + (1/2*v)*x^2*y^2 + 1/4*y^4
sage: R.<x>=L[]
sage: LL.<w> = L.extension(x^2 + v)
sage: R.<x,y> = LL[]
sage: Tetra.reynolds_operator(x^4, chi)
Traceback (most recent call last):
...
NotImplementedError: only implemented for absolute fields
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> K = CyclotomicField(Integer(4), names=('i',)); (i,) = K._first_ngens(1)
>>> Tetra =  MatrixGroup([(-Integer(1)+i)/Integer(2),(-Integer(1)+i)/
↪Integer(2), (Integer(1)+i)/Integer(2),(-Integer(1)-i)/Integer(2)],
↪[Integer(0),i, -i,Integer(0)])
>>> chi = Tetra.character(Tetra.character_table()[Integer(4)])
>>> L = QuadraticField(-Integer(3), names=('v',)); (v,) = L._first_ngens(1)
>>> R = L['x, y']; (x, y,) = R._first_ngens(2)
>>> Tetra.reynolds_operator(x**Integer(4))
0
>>> Tetra.reynolds_operator(x**Integer(4), chi)
1/4*x^4 + (1/2*v)*x^2*y^2 + 1/4*y^4
>>> R = L['x']; (x,) = R._first_ngens(1)
>>> LL = L.extension(x**Integer(2) + v, names=('w',)); (w,) = LL._first_
↪ngens(1)
>>> R = LL['x, y']; (x, y,) = R._first_ngens(2)
>>> Tetra.reynolds_operator(x**Integer(4), chi)
Traceback (most recent call last):
...
NotImplementedError: only implemented for absolute fields
```

```
sage: # needs sage.groups sage.rings.number_field
sage: G =  MatrixGroup(DihedralGroup(4))
sage: chi = G.character(G.character_table()[1])
sage: R.<x,y> = QQ[]
sage: f = x^4
sage: G.reynolds_operator(f, chi)
Traceback (most recent call last):
...
TypeError: number of variables in polynomial must match size of matrices
sage: R.<x,y,z,w> = QQ[]
sage: f = x^3*y
sage: G.reynolds_operator(f, chi)
1/8*x^3*y - 1/8*x*y^3 + 1/8*y^3*z - 1/8*y*z^3 - 1/8*x^3*w + 1/8*z^3*w +
1/8*x*w^3 - 1/8*z*w^3
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> G =  MatrixGroup(DihedralGroup(Integer(4)))
>>> chi = G.character(G.character_table()[Integer(1)])
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> f = x**Integer(4)
>>> G.reynolds_operator(f, chi)
Traceback (most recent call last):
...
TypeError: number of variables in polynomial must match size of matrices
>>> R = QQ['x, y, z, w']; (x, y, z, w,) = R._first_ngens(4)
>>> f = x**Integer(3)*y
>>> G.reynolds_operator(f, chi)
1/8*x^3*y - 1/8*x*y^3 + 1/8*y^3*z - 1/8*y*z^3 - 1/8*x^3*w + 1/8*z^3*w +
1/8*x*w^3 - 1/8*z*w^3
```

Characteristic $p > 0$ examples:

```
sage: G = MatrixGroup([[0,1, 1,0]])
sage: R.<w,x> = GF(2)[]
sage: G.reynolds_operator(x)
Traceback (most recent call last):
...
NotImplementedError: not implemented when characteristic divides group order
```

```
>>> from sage.all import *
>>> G = MatrixGroup([[Integer(0),Integer(1), Integer(1),Integer(0)]])
>>> R = GF(Integer(2))['w, x']; (w, x,) = R._first_ngens(2)
>>> G.reynolds_operator(x)
Traceback (most recent call last):
...
NotImplementedError: not implemented when characteristic divides group order
```

```
sage: i = GF(7)(3)
sage: G = MatrixGroup([[i^3,0, 0,-i^3], [i^2,0, 0,-i^2]])
sage: chi = G.character(G.character_table()[4])                          #␣
↪needs sage.rings.number_field
sage: R.<w,x> = GF(7)[]
sage: f = w^5*x + x^6
sage: G.reynolds_operator(f, chi)                                        #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
NotImplementedError: nontrivial characters not implemented for characteristic␣
↪> 0
sage: G.reynolds_operator(f)
x^6
```

```
>>> from sage.all import *
>>> i = GF(Integer(7))(Integer(3))
>>> G = MatrixGroup([[i**Integer(3),Integer(0), Integer(0),-i**Integer(3)],␣
↪[i**Integer(2),Integer(0), Integer(0),-i**Integer(2)]])
>>> chi = G.character(G.character_table()[Integer(4)])                     ␣
↪     # needs sage.rings.number_field
>>> R = GF(Integer(7))['w, x']; (w, x,) = R._first_ngens(2)
>>> f = w**Integer(5)*x + x**Integer(6)
>>> G.reynolds_operator(f, chi)                                         #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
NotImplementedError: nontrivial characters not implemented for characteristic␣
↪> 0
>>> G.reynolds_operator(f)
x^6
```

```
sage: # needs sage.rings.finite_rings
sage: K = GF(3^2,'t')
sage: G = MatrixGroup([matrix(K, 2, 2, [0,K.gen(), 1,0])])
```

(continues on next page)

(continued from previous page)

```
sage: R.<x,y> = GF(3)[]
sage: G.reynolds_operator(x^8)
-x^8 - y^8
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> K = GF(Integer(3)**Integer(2),'t')
>>> G = MatrixGroup([matrix(K, Integer(2), Integer(2), [Integer(0),K.gen(),
↪Integer(1),Integer(0)])])
>>> R = GF(Integer(3))['x, y']; (x, y,) = R._first_ngens(2)
>>> G.reynolds_operator(x**Integer(8))
-x^8 - y^8
```

```
sage: # needs sage.rings.finite_rings
sage: K = GF(3^2,'t')
sage: G = MatrixGroup([matrix(GF(3), 2, 2, [0,1, 1,0])])
sage: R.<x,y> = K[]
sage: f = -K.gen()*x
sage: G.reynolds_operator(f)
t*x + t*y
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> K = GF(Integer(3)**Integer(2),'t')
>>> G = MatrixGroup([matrix(GF(Integer(3)), Integer(2), Integer(2),
↪[Integer(0),Integer(1), Integer(1),Integer(0)])])
>>> R = K['x, y']; (x, y,) = R._first_ngens(2)
>>> f = -K.gen()*x
>>> G.reynolds_operator(f)
t*x + t*y
```

28.8 Binary Dihedral Groups

AUTHORS:

- Travis Scrimshaw (2016-02): initial version

class sage.groups.matrix_gps.binary_dihedral.**BinaryDihedralGroup**(*n*)

Bases: UniqueRepresentation, *FinitelyGeneratedMatrixGroup_gap*

The binary dihedral group BD_n of order $4n$.

Let n be a positive integer. The binary dihedral group BD_n is a finite group of order $4n$, and can be considered as the matrix group generated by

$$g_1 = \begin{pmatrix} \zeta_{2n} & 0 \\ 0 & \zeta_{2n}^{-1} \end{pmatrix}, \qquad g_2 = \begin{pmatrix} 0 & \zeta_4 \\ \zeta_4 & 0 \end{pmatrix},$$

where $\zeta_k = e^{2\pi i/k}$ is the primitive k-th root of unity. Furthermore, BD_n admits the following presentation (note that there is a typo in [Sun2010]):

$$BD_n = \langle x, y, z | x^2 = y^2 = z^n = xyz \rangle.$$

(The x, y and z in this presentations correspond to the g_2, $g_2 g_1^{-1}$ and g_1 in the matrix group avatar.)

REFERENCES:

- [Dol2009]
- [Sun2010]
- Wikipedia article Dicyclic_group#Binary_dihedral_group

cardinality()

Return the order of `self`, which is $4n$.

EXAMPLES:

```
sage: G = groups.matrix.BinaryDihedral(3)
sage: G.order()
12
```

```
>>> from sage.all import *
>>> G = groups.matrix.BinaryDihedral(Integer(3))
>>> G.order()
12
```

order()

Return the order of `self`, which is $4n$.

EXAMPLES:

```
sage: G = groups.matrix.BinaryDihedral(3)
sage: G.order()
12
```

```
>>> from sage.all import *
>>> G = groups.matrix.BinaryDihedral(Integer(3))
>>> G.order()
12
```

28.9 Coxeter Groups As Matrix Groups

This implements a general Coxeter group as a matrix group by using the reflection representation.

AUTHORS:

- Travis Scrimshaw (2013-08-28): Initial version

class sage.groups.matrix_gps.coxeter_group.**CoxeterMatrixGroup**(*coxeter_matrix*, *base_ring*, *index_set*)

Bases: `UniqueRepresentation`, *`FinitelyGeneratedMatrixGroup_generic`*

A Coxeter group represented as a matrix group.

Let (W, S) be a Coxeter system. We construct a vector space V over $\mathbf{R}$ with a basis of $\{\alpha_s\}_{s \in S}$ and inner product

$$B(\alpha_s, \alpha_t) = -\cos\left(\frac{\pi}{m_{st}}\right)$$

where we have $B(\alpha_s, \alpha_t) = -1$ if $m_{st} = \infty$. Next we define a representation $\sigma_s : V \to V$ by

$$\sigma_s \lambda = \lambda - 2B(\alpha_s, \lambda)\alpha_s.$$

This representation is faithful so we can represent the Coxeter group W by the set of matrices σ_s acting on V.

INPUT:

- `data` – a Coxeter matrix or graph or a Cartan type
- `base_ring` – (default: the universal cyclotomic field or a number field) the base ring which contains all values $\cos(\pi/m_{ij})$ where $(m_{ij})_{ij}$ is the Coxeter matrix
- `index_set` – (optional) an indexing set for the generators

For finite Coxeter groups, the default base ring is taken to be **Q** or a quadratic number field when possible.

For more on creating Coxeter groups, see `CoxeterGroup()`.

Todo

Currently the label ∞ is implemented as -1 in the Coxeter matrix.

EXAMPLES:

We can create Coxeter groups from Coxeter matrices:

```
sage: # needs sage.libs.gap sage.rings.number_field
sage: W = CoxeterGroup([[1, 6, 3], [6, 1, 10], [3, 10, 1]]); W
Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[ 1  6  3]
[ 6  1 10]
[ 3 10  1]
sage: W.gens()
(
[                 -1 -E(12)^7 + E(12)^11                   1]
[                  0                   1                   0]
[                  0                   0                   1],

[                  1                   0                   0]
[-E(12)^7 + E(12)^11                  -1     E(20) - E(20)^9]
[                  0                   0                   1],

[               1                0                0]
[               0                1                0]
[               1 E(20) - E(20)^9               -1]
)

sage: m = matrix([[1,3,3,3], [3,1,3,2], [3,3,1,2], [3,2,2,1]])
sage: W = CoxeterGroup(m)
sage: W.gens()
(
[-1  1  1  1]  [ 1  0  0  0]  [ 1  0  0  0]  [ 1  0  0  0]
[ 0  1  0  0]  [ 1 -1  1  0]  [ 0  1  0  0]  [ 0  1  0  0]
[ 0  0  1  0]  [ 0  0  1  0]  [ 1  1 -1  0]  [ 0  0  1  0]
[ 0  0  0  1], [ 0  0  0  1], [ 0  0  0  1], [ 1  0  0 -1]
```

(continues on next page)

(continued from previous page)

```
)
sage: a,b,c,d = W.gens()
sage: (a*b*c)^3
[ 5  1 -5  7]
[ 5  0 -4  5]
[ 4  1 -4  4]
[ 0  0  0  1]
sage: (a*b)^3
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: b*d == d*b
True
sage: a*c*a == c*a*c
True
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap sage.rings.number_field
>>> W = CoxeterGroup([[Integer(1), Integer(6), Integer(3)], [Integer(6),␣
↪Integer(1), Integer(10)], [Integer(3), Integer(10), Integer(1)]]); W
Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[ 1  6  3]
[ 6  1 10]
[ 3 10  1]
>>> W.gens()
(
[                 -1 -E(12)^7 + E(12)^11                  1]
[                  0                   1                  0]
[                  0                   0                  1],
<BLANKLINE>
[                  1                   0                  0]
[-E(12)^7 + E(12)^11                  -1     E(20) - E(20)^9]
[                  0                   0                  1],
<BLANKLINE>
[              1               0               0]
[              0               1               0]
[              1 E(20) - E(20)^9              -1]
)

>>> m = matrix([[Integer(1),Integer(3),Integer(3),Integer(3)], [Integer(3),
↪Integer(1),Integer(3),Integer(2)], [Integer(3),Integer(3),Integer(1),
↪Integer(2)], [Integer(3),Integer(2),Integer(2),Integer(1)]])
>>> W = CoxeterGroup(m)
>>> W.gens()
(
[-1  1  1  1]  [ 1  0  0  0]  [ 1  0  0  0]  [ 1  0  0  0]
[ 0  1  0  0]  [ 1 -1  1  0]  [ 0  1  0  0]  [ 0  1  0  0]
[ 0  0  1  0]  [ 0  0  1  0]  [ 1  1 -1  0]  [ 0  0  1  0]
[ 0  0  0  1], [ 0  0  0  1], [ 0  0  0  1], [ 1  0  0 -1]
)
>>> a,b,c,d = W.gens()
```

(continues on next page)

(continued from previous page)

```
>>> (a*b*c)**Integer(3)
[ 5  1 -5  7]
[ 5  0 -4  5]
[ 4  1 -4  4]
[ 0  0  0  1]
>>> (a*b)**Integer(3)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
>>> b*d == d*b
True
>>> a*c*a == c*a*c
True
```

We can create the matrix representation over different base rings and with different index sets. Note that the base ring must contain all $2 * \cos(\pi/m_{ij})$ where $(m_{ij})_{ij}$ is the Coxeter matrix:

```
sage: W = CoxeterGroup(m, base_ring=RR, index_set=['a','b','c','d'])
sage: W.base_ring()
Real Field with 53 bits of precision
sage: W.index_set()
('a', 'b', 'c', 'd')

sage: CoxeterGroup(m, base_ring=ZZ)
Coxeter group over Integer Ring with Coxeter matrix:
[1 3 3 3]
[3 1 3 2]
[3 3 1 2]
[3 2 2 1]
sage: CoxeterGroup([[1,4],[4,1]], base_ring=QQ)                              #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
TypeError: unable to convert sqrt(2) to a rational
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(m, base_ring=RR, index_set=['a','b','c','d'])
>>> W.base_ring()
Real Field with 53 bits of precision
>>> W.index_set()
('a', 'b', 'c', 'd')

>>> CoxeterGroup(m, base_ring=ZZ)
Coxeter group over Integer Ring with Coxeter matrix:
[1 3 3 3]
[3 1 3 2]
[3 3 1 2]
[3 2 2 1]
>>> CoxeterGroup([[Integer(1),Integer(4)],[Integer(4),Integer(1)]], base_ring=QQ)␣
↪                                  # needs sage.symbolic
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: unable to convert sqrt(2) to a rational
```

Using the well-known conversion between Coxeter matrices and Coxeter graphs, we can input a Coxeter graph. Following the standard convention, edges with no label (i.e. labelled by `None`) are treated as 3:

```
sage: # needs sage.libs.gap sage.rings.number_field
sage: G = Graph([(0,3,None), (1,3,15), (2,3,7), (0,1,3)])
sage: W = CoxeterGroup(G); W
Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[ 1  3  2  3]
[ 3  1  2 15]
[ 2  2  1  7]
[ 3 15  7  1]
sage: G2 = W.coxeter_diagram()
sage: CoxeterGroup(G2) is W
True
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap sage.rings.number_field
>>> G = Graph([(Integer(0),Integer(3),None), (Integer(1),Integer(3),Integer(15)),␣
↪(Integer(2),Integer(3),Integer(7)), (Integer(0),Integer(1),Integer(3))])
>>> W = CoxeterGroup(G); W
Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[ 1  3  2  3]
[ 3  1  2 15]
[ 2  2  1  7]
[ 3 15  7  1]
>>> G2 = W.coxeter_diagram()
>>> CoxeterGroup(G2) is W
True
```

Because there currently is no class for $\mathbf{Z} \cup \{\infty\}$, labels of ∞ are given by -1 in the Coxeter matrix:

```
sage: # needs sage.libs.gap sage.rings.number_field
sage: G = Graph([(0,1,None), (1,2,4), (0,2,oo)])
sage: W = CoxeterGroup(G)
sage: W.coxeter_matrix()
[ 1  3 -1]
[ 3  1  4]
[-1  4  1]
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap sage.rings.number_field
>>> G = Graph([(Integer(0),Integer(1),None), (Integer(1),Integer(2),Integer(4)),␣
↪(Integer(0),Integer(2),oo)])
>>> W = CoxeterGroup(G)
>>> W.coxeter_matrix()
[ 1  3 -1]
[ 3  1  4]
[-1  4  1]
```

We can also create Coxeter groups from Cartan types using the `implementation` keyword:

```
sage: W = CoxeterGroup(['D',5], implementation='reflection'); W
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3 2 2 2]
[3 1 3 2 2]
[2 3 1 3 3]
[2 2 3 1 2]
[2 2 3 2 1]
sage: W = CoxeterGroup(['H',3], implementation='reflection'); W            #␣
↪needs sage.libs.gap sage.rings.number_field
Finite Coxeter group over
 Number Field in a with defining polynomial x^2 - 5 with a = 2.236067977499790?
 with Coxeter matrix:
 [1 3 2]
 [3 1 5]
 [2 5 1]
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['D',Integer(5)], implementation='reflection'); W
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3 2 2 2]
[3 1 3 2 2]
[2 3 1 3 3]
[2 2 3 1 2]
[2 2 3 2 1]
>>> W = CoxeterGroup(['H',Integer(3)], implementation='reflection'); W     ␣
↪    # needs sage.libs.gap sage.rings.number_field
Finite Coxeter group over
 Number Field in a with defining polynomial x^2 - 5 with a = 2.236067977499790?
 with Coxeter matrix:
 [1 3 2]
 [3 1 5]
 [2 5 1]
```

class Element

Bases: *MatrixGroupElement_generic*

A Coxeter group element.

action_on_root_indices(*i*, *side='left'*)

Return the action on the set of roots.

The roots are ordered as in the output of the method `roots()`.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], implementation='reflection')
sage: w = W.w0
sage: w.action_on_root_indices(0)
11
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(3)], implementation='reflection')
>>> w = W.w0
```

(continues on next page)

(continued from previous page)

```
>>> w.action_on_root_indices(Integer(0))
11
```

canonical_matrix()

Return the matrix of `self` in the canonical faithful representation, which is `self` as a matrix.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], implementation='reflection')
sage: a,b,c = W.gens()
sage: elt = a*b*c
sage: elt.canonical_matrix()
[ 0  0 -1]
[ 1  0 -1]
[ 0  1 -1]
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(3)], implementation='reflection')
>>> a,b,c = W.gens()
>>> elt = a*b*c
>>> elt.canonical_matrix()
[ 0  0 -1]
[ 1  0 -1]
[ 0  1 -1]
```

descents(*side='right'*, *index_set=None*, *positive=False*)

Return the descents of `self`, as a list of elements of the `index_set`.

INPUT:

- `index_set` – (default: all of them) a subset (as a list or iterable) of the nodes of the Dynkin diagram
- `side` – (default: `'right'`) `'left'` or `'right'`
- `positive` – boolean (default: `False`)

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], implementation='reflection')
sage: a,b,c = W.gens()
sage: elt = b*a*c
sage: elt.descents()
[1, 3]
sage: elt.descents(positive=True)
[2]
sage: elt.descents(index_set=[1,2])
[1]
sage: elt.descents(side='left')
[2]
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(3)], implementation='reflection')
>>> a,b,c = W.gens()
>>> elt = b*a*c
>>> elt.descents()
[1, 3]
```

(continues on next page)

(continued from previous page)

```
>>> elt.descents(positive=True)
[2]
>>> elt.descents(index_set=[Integer(1),Integer(2)])
[1]
>>> elt.descents(side='left')
[2]
```

first_descent (*side='right'*, *index_set=None*, *positive=False*)

Return the first left (resp. right) descent of `self`, as ane element of `index_set`, or `None` if there is none.

See `descents()` for a description of the options.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], implementation='reflection')
sage: a,b,c = W.gens()
sage: elt = b*a*c
sage: elt.first_descent()
1
sage: elt.first_descent(side='left')
2
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(3)], implementation='reflection')
>>> a,b,c = W.gens()
>>> elt = b*a*c
>>> elt.first_descent()
1
>>> elt.first_descent(side='left')
2
```

has_right_descent (*i*)

Return whether `i` is a right descent of `self`.

A Coxeter system (W, S) has a root system defined as $\{w(\alpha_s)\}_{w\in W}$ and we define the positive (resp. negative) roots $\alpha = \sum_{s\in S} c_s\alpha_s$ by all $c_s \geq 0$ (resp. $c_s \leq 0$). In particular, we note that if $\ell(ws) > \ell(w)$ then $w(\alpha_s) > 0$ and if $\ell(ws) < \ell(w)$ then $w(\alpha_s) < 0$. Thus $i \in I$ is a right descent if $w(\alpha_{s_i}) < 0$ or equivalently if the matrix representing w has all entries of the i-th column being nonpositive.

INPUT:

- `i` – an element in the index set

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], implementation='reflection')
sage: a,b,c = W.gens()
sage: elt = b*a*c
sage: [elt.has_right_descent(i) for i in [1, 2, 3]]
[True, False, True]
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(3)], implementation='reflection')
>>> a,b,c = W.gens()
```

(continues on next page)

(continued from previous page)

```
>>> elt = b*a*c
>>> [elt.has_right_descent(i) for i in [Integer(1), Integer(2),␣
↪Integer(3)]]
[True, False, True]
```

bilinear_form()

Return the bilinear form associated to `self`.

Given a Coxeter group G with Coxeter matrix $M = (m_{ij})_{ij}$, the associated bilinear form $A = (a_{ij})_{ij}$ is given by

$$a_{ij} = -\cos\left(\frac{\pi}{m_{ij}}\right).$$

If A is positive definite, then G is of finite type (and so the associated Coxeter group is a finite group). If A is positive semidefinite, then G is affine type.

EXAMPLES:

```
sage: W = CoxeterGroup(['D',4])
sage: W.bilinear_form()                                              #␣
↪needs sage.symbolic
[   1 -1/2    0    0]
[-1/2    1 -1/2 -1/2]
[   0 -1/2    1    0]
[   0 -1/2    0    1]
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['D',Integer(4)])
>>> W.bilinear_form()                                                #␣
↪needs sage.symbolic
[   1 -1/2    0    0]
[-1/2    1 -1/2 -1/2]
[   0 -1/2    1    0]
[   0 -1/2    0    1]
```

canonical_representation()

Return the canonical faithful representation of `self`, which is `self`.

EXAMPLES:

```
sage: W = CoxeterGroup([[1,3],[3,1]])
sage: W.canonical_representation() is W
True
```

```
>>> from sage.all import *
>>> W = CoxeterGroup([[Integer(1),Integer(3)],[Integer(3),Integer(1)]])
>>> W.canonical_representation() is W
True
```

coxeter_matrix()

Return the Coxeter matrix of `self`.

EXAMPLES:

```
sage: W = CoxeterGroup([[1,3],[3,1]])
sage: W.coxeter_matrix()
[1 3]
[3 1]
sage: W = CoxeterGroup(['H',3])                                              #␣
↪needs sage.libs.gap sage.rings.number_field
sage: W.coxeter_matrix()                                                     #␣
↪needs sage.libs.gap sage.rings.number_field
[1 3 2]
[3 1 5]
[2 5 1]
```

```
>>> from sage.all import *
>>> W = CoxeterGroup([[Integer(1),Integer(3)],[Integer(3),Integer(1)]])
>>> W.coxeter_matrix()
[1 3]
[3 1]
>>> W = CoxeterGroup(['H',Integer(3)])                                       ␣
↪      # needs sage.libs.gap sage.rings.number_field
>>> W.coxeter_matrix()                                                     #␣
↪needs sage.libs.gap sage.rings.number_field
[1 3 2]
[3 1 5]
[2 5 1]
```

fundamental_weight(*i*)

Return the fundamental weight with index `i`.

See also

fundamental_weights()

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], implementation='reflection')
sage: W.fundamental_weight(1)                                                #␣
↪needs sage.symbolic
(3/2, 1, 1/2)
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(3)], implementation='reflection')
>>> W.fundamental_weight(Integer(1))                                          ␣
↪      # needs sage.symbolic
(3/2, 1, 1/2)
```

fundamental_weights()

Return the fundamental weights for `self`.

This is the dual basis to the basis of simple roots.

The base ring must be a field.

See also

fundamental_weight()

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], implementation='reflection')
sage: W.fundamental_weights()                                           #␣
↪needs sage.symbolic
Finite family {1: (3/2, 1, 1/2), 2: (1, 2, 1), 3: (1/2, 1, 3/2)}
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(3)], implementation='reflection')
>>> W.fundamental_weights()                                            #␣
↪needs sage.symbolic
Finite family {1: (3/2, 1, 1/2), 2: (1, 2, 1), 3: (1/2, 1, 3/2)}
```

is_commutative()

Return whether `self` is commutative.

EXAMPLES:

```
sage: CoxeterGroup(['A', 2]).is_commutative()
False
sage: W = CoxeterGroup(['I',2])
sage: W.is_commutative()
True
```

```
>>> from sage.all import *
>>> CoxeterGroup(['A', Integer(2)]).is_commutative()
False
>>> W = CoxeterGroup(['I',Integer(2)])
>>> W.is_commutative()
True
```

is_finite()

Return `True` if this group is finite.

EXAMPLES:

```
sage: # needs sage.libs.gap sage.rings.number_field
sage: [l for l in range(2, 9) if
....:  CoxeterGroup([[1,3,2],[3,1,l],[2,l,1]]).is_finite()]
[2, 3, 4, 5]
sage: [l for l in range(2, 9) if
....:  CoxeterGroup([[1,3,2,2],[3,1,l,2],[2,l,1,3],[2,2,3,1]]).is_finite()]
[2, 3, 4]
sage: [l for l in range(2, 9) if
....:  CoxeterGroup([[1,3,2,2,2], [3,1,3,3,2], [2,3,1,2,2],
....:                [2,3,2,1,l], [2,2,2,l,1]]).is_finite()]
[2, 3]
sage: [l for l in range(2, 9) if
....:  CoxeterGroup([[1,3,2,2,2], [3,1,2,3,3], [2,2,1,l,2],
```

(continues on next page)

(continued from previous page)

```
....:                   [2,3,l,1,2], [2,3,2,2,1]]).is_finite()]
[2, 3]
sage: [l for l in range(2, 9) if
....:  CoxeterGroup([[1,3,2,2,2,2], [3,1,l,2,2,2], [2,l,1,3,l,2],
....:                   [2,2,3,1,2,2], [2,2,l,2,1,3], [2,2,2,2,3,1]]).is_
↪finite()]
[2, 3]
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap sage.rings.number_field
>>> [l for l in range(Integer(2), Integer(9)) if
...  CoxeterGroup([[Integer(1),Integer(3),Integer(2)],[Integer(3),Integer(1),
↪l],[Integer(2),l,Integer(1)]]).is_finite()]
[2, 3, 4, 5]
>>> [l for l in range(Integer(2), Integer(9)) if
...  CoxeterGroup([[Integer(1),Integer(3),Integer(2),Integer(2)],[Integer(3),
↪Integer(1),l,Integer(2)],[Integer(2),l,Integer(1),Integer(3)],[Integer(2),
↪Integer(2),Integer(3),Integer(1)]]).is_finite()]
[2, 3, 4]
>>> [l for l in range(Integer(2), Integer(9)) if
...  CoxeterGroup([[Integer(1),Integer(3),Integer(2),Integer(2),Integer(2)],␣
↪[Integer(3),Integer(1),Integer(3),Integer(3),Integer(2)], [Integer(2),
↪Integer(3),Integer(1),Integer(2),Integer(2)],
...                   [Integer(2),Integer(3),Integer(2),Integer(1),l],␣
↪[Integer(2),Integer(2),Integer(2),l,Integer(1)]]).is_finite()]
[2, 3]
>>> [l for l in range(Integer(2), Integer(9)) if
...  CoxeterGroup([[Integer(1),Integer(3),Integer(2),Integer(2),Integer(2)],␣
↪[Integer(3),Integer(1),Integer(2),Integer(3),Integer(3)], [Integer(2),
↪Integer(2),Integer(1),l,Integer(2)],
...                   [Integer(2),Integer(3),l,Integer(1),Integer(2)],␣
↪[Integer(2),Integer(3),Integer(2),Integer(2),Integer(1)]]).is_finite()]
[2, 3]
>>> [l for l in range(Integer(2), Integer(9)) if
...  CoxeterGroup([[Integer(1),Integer(3),Integer(2),Integer(2),Integer(2),
↪Integer(2)], [Integer(3),Integer(1),l,Integer(2),Integer(2),Integer(2)],␣
↪[Integer(2),l,Integer(1),Integer(3),l,Integer(2)],
...                   [Integer(2),Integer(2),Integer(3),Integer(1),Integer(2),
↪Integer(2)], [Integer(2),Integer(2),l,Integer(2),Integer(1),Integer(3)],␣
↪[Integer(2),Integer(2),Integer(2),Integer(2),Integer(3),Integer(1)]]).is_
↪finite()]
[2, 3]
```

order()

Return the order of `self`.

If the Coxeter group is finite, this uses an iterator.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: W = CoxeterGroup([[1,3],[3,1]])
sage: W.order()
```

(continues on next page)

(continued from previous page)

```
6
sage: W = CoxeterGroup([[1,-1],[-1,1]])                                 #␣
↪needs sage.libs.gap
sage: W.order()                                                         #␣
↪needs sage.libs.gap
+Infinity
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> W = CoxeterGroup([[Integer(1),Integer(3)],[Integer(3),Integer(1)]])
>>> W.order()
6
>>> W = CoxeterGroup([[Integer(1),-Integer(1)],[-Integer(1),Integer(1)]])    ␣
↪                                # needs sage.libs.gap
>>> W.order()                                                          #␣
↪needs sage.libs.gap
+Infinity
```

positive_roots()

Return the positive roots.

These are roots in the Coxeter sense, that all have the same norm. They are given by their coefficients in the base of simple roots, also taken to have all the same norm.

> **See also**
>
> *reflections()*

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], implementation='reflection')
sage: W.positive_roots()
((1, 0, 0), (1, 1, 0), (0, 1, 0), (1, 1, 1), (0, 1, 1), (0, 0, 1))

sage: # needs sage.libs.gap sage.rings.number_field
sage: W = CoxeterGroup(['I',5], implementation='reflection')
sage: W.positive_roots()
((1, 0),
 (-E(5)^2 - E(5)^3, 1),
 (-E(5)^2 - E(5)^3, -E(5)^2 - E(5)^3),
 (1, -E(5)^2 - E(5)^3),
 (0, 1))
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(3)], implementation='reflection')
>>> W.positive_roots()
((1, 0, 0), (1, 1, 0), (0, 1, 0), (1, 1, 1), (0, 1, 1), (0, 0, 1))

>>> # needs sage.libs.gap sage.rings.number_field
>>> W = CoxeterGroup(['I',Integer(5)], implementation='reflection')
>>> W.positive_roots()
```

(continues on next page)

(continued from previous page)

```
((1, 0),
 (-E(5)^2 - E(5)^3, 1),
 (-E(5)^2 - E(5)^3, -E(5)^2 - E(5)^3),
 (1, -E(5)^2 - E(5)^3),
 (0, 1))
```

reflections()

Return the set of reflections.

The order is the one given by *positive_roots()*.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], implementation='reflection')
sage: list(W.reflections())
[
[-1  1]  [ 0 -1]  [ 1  0]
[ 0  1], [-1  0], [ 1 -1]
]
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(2)], implementation='reflection')
>>> list(W.reflections())
[
[-1  1]  [ 0 -1]  [ 1  0]
[ 0  1], [-1  0], [ 1 -1]
]
```

roots()

Return the roots.

These are roots in the Coxeter sense, that all have the same norm. They are given by their coefficients in the base of simple roots, also taken to have all the same norm.

The positive roots are listed first, then the negative roots in the same order. The order is the one given by *roots()*.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], implementation='reflection')
sage: W.roots()
((1, 0, 0),
 (1, 1, 0),
 (0, 1, 0),
 (1, 1, 1),
 (0, 1, 1),
 (0, 0, 1),
 (-1, 0, 0),
 (-1, -1, 0),
 (0, -1, 0),
 (-1, -1, -1),
 (0, -1, -1),
 (0, 0, -1))
```

(continues on next page)

(continued from previous page)

```
sage: # needs sage.libs.gap sage.rings.number_field
sage: W = CoxeterGroup(['I',5], implementation='reflection')
sage: len(W.roots())
10
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(3)], implementation='reflection')
>>> W.roots()
((1, 0, 0),
 (1, 1, 0),
 (0, 1, 0),
 (1, 1, 1),
 (0, 1, 1),
 (0, 0, 1),
 (-1, 0, 0),
 (-1, -1, 0),
 (0, -1, 0),
 (-1, -1, -1),
 (0, -1, -1),
 (0, 0, -1))

>>> # needs sage.libs.gap sage.rings.number_field
>>> W = CoxeterGroup(['I',Integer(5)], implementation='reflection')
>>> len(W.roots())
10
```

simple_reflection(i)

Return the simple reflection s_i.

INPUT:

- `i` – an element from the index set

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], implementation='reflection')
sage: W.simple_reflection(1)
[-1  1  0]
[ 0  1  0]
[ 0  0  1]
sage: W.simple_reflection(2)
[ 1  0  0]
[ 1 -1  1]
[ 0  0  1]
sage: W.simple_reflection(3)
[ 1  0  0]
[ 0  1  0]
[ 0  1 -1]
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(3)], implementation='reflection')
>>> W.simple_reflection(Integer(1))
[-1  1  0]
```

(continues on next page)

(continued from previous page)

```
[ 0  1  0]
[ 0  0  1]
>>> W.simple_reflection(Integer(2))
[ 1  0  0]
[ 1 -1  1]
[ 0  0  1]
>>> W.simple_reflection(Integer(3))
[ 1  0  0]
[ 0  1  0]
[ 0  1 -1]
```

simple_root_index(i)

Return the index of the simple root α_i.

This is the position of α_i in the list of all roots as given be *roots()*.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], implementation='reflection')
sage: [W.simple_root_index(i) for i in W.index_set()]
[0, 2, 5]
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(3)], implementation='reflection')
>>> [W.simple_root_index(i) for i in W.index_set()]
[0, 2, 5]
```

28.10 Linear Groups

EXAMPLES:

```
sage: GL(4, QQ)
General Linear Group of degree 4 over Rational Field
sage: GL(1, ZZ)
General Linear Group of degree 1 over Integer Ring
sage: GL(100, RR)
General Linear Group of degree 100 over Real Field with 53 bits of precision
sage: GL(3, GF(49,'a'))                                                   #␣
↪needs sage.rings.finite_rings
General Linear Group of degree 3 over Finite Field in a of size 7^2

sage: SL(2, ZZ)
Special Linear Group of degree 2 over Integer Ring
sage: G = SL(2, GF(3)); G
Special Linear Group of degree 2 over Finite Field of size 3

sage: # needs sage.libs.gap
sage: G.is_finite()
True
sage: G.conjugacy_classes_representatives()
(
[1 0]  [0 2]  [0 1]  [2 0]  [0 2]  [0 1]  [0 2]
```

(continues on next page)

(continued from previous page)

```
[0 1], [1 1], [2 1], [0 2], [1 2], [2 2], [1 0]
)
sage: G = SL(6, GF(5))
sage: G.gens()
(
[2 0 0 0 0 0]  [4 0 0 0 0 1]
[0 3 0 0 0 0]  [4 0 0 0 0 0]
[0 0 1 0 0 0]  [0 4 0 0 0 0]
[0 0 0 1 0 0]  [0 0 4 0 0 0]
[0 0 0 0 1 0]  [0 0 0 4 0 0]
[0 0 0 0 0 1], [0 0 0 0 4 0]
)
```

```
>>> from sage.all import *
>>> GL(Integer(4), QQ)
General Linear Group of degree 4 over Rational Field
>>> GL(Integer(1), ZZ)
General Linear Group of degree 1 over Integer Ring
>>> GL(Integer(100), RR)
General Linear Group of degree 100 over Real Field with 53 bits of precision
>>> GL(Integer(3), GF(Integer(49),'a'))
↪                # needs sage.rings.finite_rings
General Linear Group of degree 3 over Finite Field in a of size 7^2

>>> SL(Integer(2), ZZ)
Special Linear Group of degree 2 over Integer Ring
>>> G = SL(Integer(2), GF(Integer(3))); G
Special Linear Group of degree 2 over Finite Field of size 3

>>> # needs sage.libs.gap
>>> G.is_finite()
True
>>> G.conjugacy_classes_representatives()
(
[1 0]  [0 2]  [0 1]  [2 0]  [0 2]  [0 1]  [0 2]
[0 1], [1 1], [2 1], [0 2], [1 2], [2 2], [1 0]
)
>>> G = SL(Integer(6), GF(Integer(5)))
>>> G.gens()
(
[2 0 0 0 0 0]  [4 0 0 0 0 1]
[0 3 0 0 0 0]  [4 0 0 0 0 0]
[0 0 1 0 0 0]  [0 4 0 0 0 0]
[0 0 0 1 0 0]  [0 0 4 0 0 0]
[0 0 0 0 1 0]  [0 0 0 4 0 0]
[0 0 0 0 0 1], [0 0 0 0 4 0]
)
```

AUTHORS:

- William Stein: initial version
- David Joyner: degree, base_ring, random, order methods; examples

- David Joyner (2006-05): added center, more examples, renamed random attributes, bug fixes.
- William Stein (2006-12): total rewrite
- Volker Braun (2013-1) port to new Parent, libGAP, extreme refactoring.

REFERENCES: See [KL1990] and [Car1972].

`sage.groups.matrix_gps.linear.`**`GL`**`(`*n*, *R*, *var='a'*`)`

Return the general linear group.

The general linear group $GL(d, R)$ consists of all $d \times d$ matrices that are invertible over the ring R.

> **Note**
>
> This group is also available via `groups.matrix.GL()`.

INPUT:

- `n` – positive integer
- `R` – ring or an integer; if an integer is specified, the corresponding finite field is used
- `var` – variable used to represent generator of the finite field, if needed

EXAMPLES:

```
sage: G = GL(6, GF(5))
sage: G.base_ring()
Finite Field of size 5
sage: G.category()
Category of finite groups

sage: # needs sage.libs.gap
sage: G.order()
11064475422000000000000000
sage: TestSuite(G).run()

sage: G = GL(6, QQ)
sage: G.category()
Category of infinite groups

sage: # needs sage.libs.gap
sage: TestSuite(G).run()
```

```
>>> from sage.all import *
>>> G = GL(Integer(6), GF(Integer(5)))
>>> G.base_ring()
Finite Field of size 5
>>> G.category()
Category of finite groups

>>> # needs sage.libs.gap
>>> G.order()
11064475422000000000000000
>>> TestSuite(G).run()
```

(continues on next page)

(continued from previous page)

```
>>> G = GL(Integer(6), QQ)
>>> G.category()
Category of infinite groups

>>> # needs sage.libs.gap
>>> TestSuite(G).run()
```

Here is the Cayley graph of (relatively small) finite General Linear Group:

```
sage: # needs sage.graphs sage.libs.gap
sage: g = GL(2,3)
sage: d = g.cayley_graph(); d
Digraph on 48 vertices
sage: d.plot(color_by_label=True, vertex_size=0.03,      # long time             #␣
↪needs sage.plot
....:        vertex_labels=False)
Graphics object consisting of 144 graphics primitives
sage: d.plot3d(color_by_label=True)                      # long time             #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.libs.gap
>>> g = GL(Integer(2),Integer(3))
>>> d = g.cayley_graph(); d
Digraph on 48 vertices
>>> d.plot(color_by_label=True, vertex_size=RealNumber('0.03'),      # long time  ␣
↪          # needs sage.plot
...        vertex_labels=False)
Graphics object consisting of 144 graphics primitives
>>> d.plot3d(color_by_label=True)                      # long time             #␣
↪needs sage.plot
Graphics3d Object
```

```
sage: # needs sage.libs.gap
sage: F = GF(3); MS = MatrixSpace(F, 2, 2)
sage: gens = [MS([[2,0], [0,1]]), MS([[2,1], [2,0]])]
sage: G = MatrixGroup(gens)
sage: G.order()
48
sage: G.cardinality()
48
sage: H = GL(2,F)
sage: H.order()
48
sage: H == G
True
sage: H.gens() == G.gens()
True
sage: H.as_matrix_group() == H
True
```

(continues on next page)

(continued from previous page)

```
sage: H.gens()
(
[2 0]  [2 1]
[0 1], [2 0]
)
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> F = GF(Integer(3)); MS = MatrixSpace(F, Integer(2), Integer(2))
>>> gens = [MS([[Integer(2),Integer(0)], [Integer(0),Integer(1)]]),
↪MS([[Integer(2),Integer(1)], [Integer(2),Integer(0)]])]
>>> G = MatrixGroup(gens)
>>> G.order()
48
>>> G.cardinality()
48
>>> H = GL(Integer(2),F)
>>> H.order()
48
>>> H == G
True
>>> H.gens() == G.gens()
True
>>> H.as_matrix_group() == H
True
>>> H.gens()
(
[2 0]  [2 1]
[0 1], [2 0]
)
```

class sage.groups.matrix_gps.linear.**LinearMatrixGroup_generic**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *category=None*, *invariant_form=None*)

Bases: *NamedMatrixGroup_generic*

cardinality()

Return the order of `self`.

EXAMPLES:

```
sage: G = SL(3, GF(5))
sage: G.order()
372000
```

```
>>> from sage.all import *
>>> G = SL(Integer(3), GF(Integer(5)))
>>> G.order()
372000
```

The order computation also works over the base rings $\mathbf{Z}/n\mathbf{Z}$:

```
sage: GL(4, Integers(15)).order()
2815842631680000000

sage: SL(4, Integers(15)).order()
351980328960000000

sage: G = GL(2, Integers(6))
sage: G.order() == len(list(G))
True

sage: H = SL(2, Integers(6))
sage: H.order() == len(list(H))
True
```

```
>>> from sage.all import *
>>> GL(Integer(4), Integers(Integer(15))).order()
2815842631680000000

>>> SL(Integer(4), Integers(Integer(15))).order()
351980328960000000

>>> G = GL(Integer(2), Integers(Integer(6)))
>>> G.order() == len(list(G))
True

>>> H = SL(Integer(2), Integers(Integer(6)))
>>> H.order() == len(list(H))
True
```

Arbitrary base rings are currently not fully supported:

```
sage: R.<x> = PolynomialRing(GF(7))
sage: S = R.quotient(x^2 + 5)
sage: GL(2, S).order()
Traceback (most recent call last):
...
NotImplementedError: order computation of linear groups not fully supported
↪for arbitrary base rings
```

```
>>> from sage.all import *
>>> R = PolynomialRing(GF(Integer(7)), names=('x',)); (x,) = R._first_ngens(1)
>>> S = R.quotient(x**Integer(2) + Integer(5))
>>> GL(Integer(2), S).order()
Traceback (most recent call last):
...
NotImplementedError: order computation of linear groups not fully supported
↪for arbitrary base rings
```

order()

Return the order of `self`.

EXAMPLES:

```
sage: G = SL(3, GF(5))
sage: G.order()
372000
```

```
>>> from sage.all import *
>>> G = SL(Integer(3), GF(Integer(5)))
>>> G.order()
372000
```

The order computation also works over the base rings $\mathbf{Z}/n\mathbf{Z}$:

```
sage: GL(4, Integers(15)).order()
2815842631680000000

sage: SL(4, Integers(15)).order()
351980328960000000

sage: G = GL(2, Integers(6))
sage: G.order() == len(list(G))
True

sage: H = SL(2, Integers(6))
sage: H.order() == len(list(H))
True
```

```
>>> from sage.all import *
>>> GL(Integer(4), Integers(Integer(15))).order()
2815842631680000000

>>> SL(Integer(4), Integers(Integer(15))).order()
351980328960000000

>>> G = GL(Integer(2), Integers(Integer(6)))
>>> G.order() == len(list(G))
True

>>> H = SL(Integer(2), Integers(Integer(6)))
>>> H.order() == len(list(H))
True
```

Arbitrary base rings are currently not fully supported:

```
sage: R.<x> = PolynomialRing(GF(7))
sage: S = R.quotient(x^2 + 5)
sage: GL(2, S).order()
Traceback (most recent call last):
...
NotImplementedError: order computation of linear groups not fully supported␣
↪for arbitrary base rings
```

```
>>> from sage.all import *
>>> R = PolynomialRing(GF(Integer(7)), names=('x',)); (x,) = R._first_ngens(1)
```

(continues on next page)

(continued from previous page)

```
>>> S = R.quotient(x**Integer(2) + Integer(5))
>>> GL(Integer(2), S).order()
Traceback (most recent call last):
...
NotImplementedError: order computation of linear groups not fully supported␣
↪for arbitrary base rings
```

sage.groups.matrix_gps.linear.**SL**(*n*, *R*, *var='a'*)

Return the special linear group.

The special linear group $SL(d, R)$ consists of all $d \times d$ matrices that are invertible over the ring R with determinant one.

> **Note**
>
> This group is also available via `groups.matrix.SL()`.

INPUT:

- `n` – positive integer
- `R` – ring or integer; if an integer is specified, the corresponding finite field is used
- `var` – variable used to represent generator of the finite field, if needed

EXAMPLES:

```
sage: SL(3, GF(2))
Special Linear Group of degree 3 over Finite Field of size 2
sage: G = SL(15, GF(7)); G
Special Linear Group of degree 15 over Finite Field of size 7
sage: G.category()
Category of finite groups

sage: # needs sage.libs.gap
sage: G.order()
1956712595698146962015219062429586341124018007182049478916067369638713066737882363
↪393519966343657677430907011270206265834819092046250232049187967718149558134226774
↪650845658791865745408000000
sage: len(G.gens())
2

sage: G = SL(2, ZZ); G
Special Linear Group of degree 2 over Integer Ring
sage: G.category()
Category of infinite groups
sage: G.gens()
(
[ 0  1]  [1 1]
[-1  0], [0 1]
)
```

```
>>> from sage.all import *
>>> SL(Integer(3), GF(Integer(2)))
Special Linear Group of degree 3 over Finite Field of size 2
>>> G = SL(Integer(15), GF(Integer(7))); G
Special Linear Group of degree 15 over Finite Field of size 7
>>> G.category()
Category of finite groups

>>> # needs sage.libs.gap
>>> G.order()
1956712595698146962015219062429586341124018007182049478916067369638713066737882363
↪393519966343657677430907011270206265834819092046250232049187967718149558134226774
↪650845658791865745408000000
>>> len(G.gens())
2

>>> G = SL(Integer(2), ZZ); G
Special Linear Group of degree 2 over Integer Ring
>>> G.category()
Category of infinite groups
>>> G.gens()
(
[ 0  1]  [1 1]
[-1  0], [0 1]
)
```

Next we compute generators for $\mathrm{SL}_3(\mathbf{Z})$

```
sage: G = SL(3, ZZ); G
Special Linear Group of degree 3 over Integer Ring

sage: # needs sage.libs.gap
sage: G.gens()
(
[0 1 0]  [ 0  1  0]  [1 1 0]
[0 0 1]  [-1  0  0]  [0 1 0]
[1 0 0], [ 0  0  1], [0 0 1]
)
sage: TestSuite(G).run()
```

```
>>> from sage.all import *
>>> G = SL(Integer(3), ZZ); G
Special Linear Group of degree 3 over Integer Ring

>>> # needs sage.libs.gap
>>> G.gens()
(
[0 1 0]  [ 0  1  0]  [1 1 0]
[0 0 1]  [-1  0  0]  [0 1 0]
[1 0 0], [ 0  0  1], [0 0 1]
)
>>> TestSuite(G).run()
```

28.11 Linear Groups with GAP

class sage.groups.matrix_gps.linear_gap.**LinearMatrixGroup_gap**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *gap_command_string*, *category=None*)

Bases: *NamedMatrixGroup_gap*, *LinearMatrixGroup_generic*, *FinitelyGeneratedMatrixGroup_gap*

The general or special linear group in GAP.

28.12 Orthogonal Linear Groups

The general orthogonal group $GO(n, R)$ consists of all $n \times n$ matrices over the ring R preserving an n-ary positive definite quadratic form. In cases where there are multiple non-isomorphic quadratic forms, additional data needs to be specified to disambiguate. The special orthogonal group is the normal subgroup of matrices of determinant one.

In characteristics different from 2, a quadratic form is equivalent to a bilinear symmetric form. Furthermore, over the real numbers a positive definite quadratic form is equivalent to the diagonal quadratic form, equivalent to the bilinear symmetric form defined by the identity matrix. Hence, the orthogonal group $GO(n, \mathbf{R})$ is the group of orthogonal matrices in the usual sense.

In the case of a finite field and if the degree n is even, then there are two inequivalent quadratic forms and a third parameter `e` must be specified to disambiguate these two possibilities. The index of $SO(e, d, q)$ in $GO(e, d, q)$ is 2 if q is odd, but $SO(e, d, q) = GO(e, d, q)$ if q is even.)

Warning

GAP and Sage use different notations:

- GAP notation: The optional `e` comes first, that is, `GO([e,] d, q)`, `SO([e,] d, q)`.
- Sage notation: The optional `e` comes last, the standard Python convention: `GO(d, GF(q), e=0)`, `SO(d, GF(q), e=0)`.

EXAMPLES:

```
sage: GO(3,7)
General Orthogonal Group of degree 3 over Finite Field of size 7

sage: G = SO(4, GF(7), 1); G
Special Orthogonal Group of degree 4 and form parameter 1
 over Finite Field of size 7

sage: # needs sage.libs.gap
sage: G.random_element()    # random
[4 3 5 2]
[6 6 4 0]
[0 4 6 0]
[4 4 5 1]
```

```
>>> from sage.all import *
>>> GO(Integer(3),Integer(7))
```

(continues on next page)

(continued from previous page)

```
General Orthogonal Group of degree 3 over Finite Field of size 7

>>> G = SO(Integer(4), GF(Integer(7)), Integer(1)); G
Special Orthogonal Group of degree 4 and form parameter 1
 over Finite Field of size 7

>>> # needs sage.libs.gap
>>> G.random_element()    # random
[4 3 5 2]
[6 6 4 0]
[0 4 6 0]
[4 4 5 1]
```

AUTHORS:

- David Joyner (2006-03): initial version
- David Joyner (2006-05): added examples, _latex_, __str__, gens, as_matrix_group
- William Stein (2006-12-09): rewrite
- Volker Braun (2013-1) port to new Parent, libGAP, extreme refactoring.
- Sebastian Oehms (2018-8) add *invariant_form()* (as alias), `_OG`, option for user defined invariant bilinear form, and bug-fix in cmd-string for calling GAP (see Issue #26028)

`sage.groups.matrix_gps.orthogonal.`**GO**(*n*, *R*, *e=0*, *var='a'*, *invariant_form=None*)

Return the general orthogonal group.

The general orthogonal group $GO(n, R)$ consists of all $n \times n$ matrices over the ring R preserving an n-ary positive definite quadratic form. In cases where there are multiple non-isomorphic quadratic forms, additional data needs to be specified to disambiguate.

In the case of a finite field and if the degree n is even, then there are two inequivalent quadratic forms and a third parameter `e` must be specified to disambiguate these two possibilities.

Note

This group is also available via `groups.matrix.GO()`.

INPUT:

- `n` – integer; the degree
- `R` – ring or an integer; if an integer is specified, the corresponding finite field is used
- `e` – `+1` or `-1`, and ignored by default; only relevant for finite fields and if the degree is even: a parameter that distinguishes inequivalent invariant forms
- `var` – (default: `'a'`) variable used to represent generator of the finite field, if needed
- `invariant_form` – (optional) instances being accepted by the matrix-constructor which define a $n \times n$ square matrix over `R` describing the symmetric form to be kept invariant by the orthogonal group; the form is checked to be non-degenerate and symmetric but not to be positive definite

OUTPUT: the general orthogonal group of given degree, base ring, and choice of invariant form

EXAMPLES:

```
sage: GO(3, GF(7))
General Orthogonal Group of degree 3 over Finite Field of size 7

sage: # needs sage.libs.gap
sage: GO(3, GF(7)).order()
672
sage: GO(3, GF(7)).gens()
(
[3 0 0]  [0 1 0]
[0 5 0]  [1 6 6]
[0 0 1], [0 2 1]
)
```

```
>>> from sage.all import *
>>> GO(Integer(3), GF(Integer(7)))
General Orthogonal Group of degree 3 over Finite Field of size 7

>>> # needs sage.libs.gap
>>> GO(Integer(3), GF(Integer(7))).order()
672
>>> GO(Integer(3), GF(Integer(7))).gens()
(
[3 0 0]  [0 1 0]
[0 5 0]  [1 6 6]
[0 0 1], [0 2 1]
)
```

Using the `invariant_form` option:

```
sage: m = matrix(QQ, 3, 3, [[0, 1, 0], [1, 0, 0], [0, 0, 3]])
sage: GO3  = GO(3, QQ)
sage: GO3m = GO(3, QQ, invariant_form=m)
sage: GO3 == GO3m
False
sage: GO3.invariant_form()
[1 0 0]
[0 1 0]
[0 0 1]
sage: GO3m.invariant_form()
[0 1 0]
[1 0 0]
[0 0 3]
sage: pm = Permutation([2,3,1]).to_matrix()
sage: g = GO3(pm); g in GO3; g
True
[0 0 1]
[1 0 0]
[0 1 0]
sage: GO3m(pm)
Traceback (most recent call last):
...
TypeError: matrix must be orthogonal with respect to the symmetric form
[0 1 0]
```

(continues on next page)

(continued from previous page)

```
[1 0 0]
[0 0 3]

sage: GO(3,3, invariant_form=[[1,0,0], [0,2,0], [0,0,1]])
Traceback (most recent call last):
...
NotImplementedError: invariant_form for finite groups is fixed by GAP
sage: 5 + 5
10
sage: R.<x> = ZZ[]
sage: GO(2, R, invariant_form=[[x,0], [0,1]])
General Orthogonal Group of degree 2 over
 Univariate Polynomial Ring in x over Integer Ring with respect to symmetric form
[x 0]
[0 1]
```

```
>>> from sage.all import *
>>> m = matrix(QQ, Integer(3), Integer(3), [[Integer(0), Integer(1), Integer(0)],␣
↪[Integer(1), Integer(0), Integer(0)], [Integer(0), Integer(0), Integer(3)]])
>>> GO3  = GO(Integer(3), QQ)
>>> GO3m = GO(Integer(3), QQ, invariant_form=m)
>>> GO3 == GO3m
False
>>> GO3.invariant_form()
[1 0 0]
[0 1 0]
[0 0 1]
>>> GO3m.invariant_form()
[0 1 0]
[1 0 0]
[0 0 3]
>>> pm = Permutation([Integer(2),Integer(3),Integer(1)]).to_matrix()
>>> g = GO3(pm); g in GO3; g
True
[0 0 1]
[1 0 0]
[0 1 0]
>>> GO3m(pm)
Traceback (most recent call last):
...
TypeError: matrix must be orthogonal with respect to the symmetric form
[0 1 0]
[1 0 0]
[0 0 3]

>>> GO(Integer(3),Integer(3), invariant_form=[[Integer(1),Integer(0),Integer(0)],␣
↪[Integer(0),Integer(2),Integer(0)], [Integer(0),Integer(0),Integer(1)]])
Traceback (most recent call last):
...
NotImplementedError: invariant_form for finite groups is fixed by GAP
>>> Integer(5) + Integer(5)
10
```

(continues on next page)

(continued from previous page)

```
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> GO(Integer(2), R, invariant_form=[[x,Integer(0)], [Integer(0),Integer(1)]])
General Orthogonal Group of degree 2 over
 Univariate Polynomial Ring in x over Integer Ring with respect to symmetric form
[x 0]
[0 1]
```

class sage.groups.matrix_gps.orthogonal.**OrthogonalMatrixGroup_generic**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *category=None*, *invariant_form=None*)

Bases: *NamedMatrixGroup_generic*

General Orthogonal Group over arbitrary rings.

EXAMPLES:

```
sage: G = GO(3, GF(7)); G
General Orthogonal Group of degree 3 over Finite Field of size 7
sage: latex(G)
\text{GO}_{3}(\Bold{F}_{7})

sage: G = SO(3, GF(5));  G
Special Orthogonal Group of degree 3 over Finite Field of size 5
sage: latex(G)
\text{SO}_{3}(\Bold{F}_{5})

sage: # needs sage.rings.number_field
sage: CF3 = CyclotomicField(3); e3 = CF3.gen()
sage: m = matrix(CF3, 3,3, [[1,e3,0],[e3,2,0],[0,0,1]])
sage: G = SO(3, CF3, invariant_form=m)
sage: latex(G)
\text{SO}_{3}(\Bold{Q}(\zeta_{3}))\text{ with respect to non positive definite␣
↪symmetric form }\left(\begin{array}{rrr}
1 & \zeta_{3} & 0 \\
\zeta_{3} & 2 & 0 \\
0 & 0 & 1
\end{array}\right)
```

```
>>> from sage.all import *
>>> G = GO(Integer(3), GF(Integer(7))); G
General Orthogonal Group of degree 3 over Finite Field of size 7
>>> latex(G)
\text{GO}_{3}(\Bold{F}_{7})

>>> G = SO(Integer(3), GF(Integer(5)));  G
Special Orthogonal Group of degree 3 over Finite Field of size 5
>>> latex(G)
\text{SO}_{3}(\Bold{F}_{5})

>>> # needs sage.rings.number_field
>>> CF3 = CyclotomicField(Integer(3)); e3 = CF3.gen()
```

(continues on next page)

(continued from previous page)

```
>>> m = matrix(CF3, Integer(3),Integer(3), [[Integer(1),e3,Integer(0)],[e3,
↪Integer(2),Integer(0)],[Integer(0),Integer(0),Integer(1)]])
>>> G = SO(Integer(3), CF3, invariant_form=m)
>>> latex(G)
\text{SO}_{3}(\Bold{Q}(\zeta_{3}))\text{ with respect to non positive definite␣
↪symmetric form }\left(\begin{array}{rrr}
1 & \zeta_{3} & 0 \\
\zeta_{3} & 2 & 0 \\
0 & 0 & 1
\end{array}\right)
```

invariant_bilinear_form()

Return the symmetric bilinear form preserved by `self`.

OUTPUT: a matrix

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: GO(2,3,+1).invariant_bilinear_form()
[0 1]
[1 0]
sage: GO(2,3,-1).invariant_bilinear_form()
[2 1]
[1 1]
sage: G = GO(4, QQ)
sage: G.invariant_bilinear_form()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: GO3m = GO(3, QQ, invariant_form=(1,0,0, 0,2,0, 0,0,3))
sage: GO3m.invariant_bilinear_form()
[1 0 0]
[0 2 0]
[0 0 3]
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> GO(Integer(2),Integer(3),+Integer(1)).invariant_bilinear_form()
[0 1]
[1 0]
>>> GO(Integer(2),Integer(3),-Integer(1)).invariant_bilinear_form()
[2 1]
[1 1]
>>> G = GO(Integer(4), QQ)
>>> G.invariant_bilinear_form()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
>>> GO3m = GO(Integer(3), QQ, invariant_form=(Integer(1),Integer(0),
↪Integer(0), Integer(0),Integer(2),Integer(0), Integer(0),Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(3)))
>>> GO3m.invariant_bilinear_form()
[1 0 0]
[0 2 0]
[0 0 3]
```

invariant_form()

Return the symmetric bilinear form preserved by `self`.

OUTPUT: a matrix

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: GO(2,3,+1).invariant_bilinear_form()
[0 1]
[1 0]
sage: GO(2,3,-1).invariant_bilinear_form()
[2 1]
[1 1]
sage: G = GO(4, QQ)
sage: G.invariant_bilinear_form()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: GO3m = GO(3, QQ, invariant_form=(1,0,0, 0,2,0, 0,0,3))
sage: GO3m.invariant_bilinear_form()
[1 0 0]
[0 2 0]
[0 0 3]
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> GO(Integer(2),Integer(3),+Integer(1)).invariant_bilinear_form()
[0 1]
[1 0]
>>> GO(Integer(2),Integer(3),-Integer(1)).invariant_bilinear_form()
[2 1]
[1 1]
>>> G = GO(Integer(4), QQ)
>>> G.invariant_bilinear_form()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
>>> GO3m = GO(Integer(3), QQ, invariant_form=(Integer(1),Integer(0),
↪Integer(0), Integer(0),Integer(2),Integer(0), Integer(0),Integer(0),
↪Integer(3)))
>>> GO3m.invariant_bilinear_form()
[1 0 0]
[0 2 0]
[0 0 3]
```

invariant_quadratic_form()

Return the symmetric bilinear form preserved by `self`.

OUTPUT: a matrix

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: GO(2,3,+1).invariant_bilinear_form()
[0 1]
[1 0]
sage: GO(2,3,-1).invariant_bilinear_form()
[2 1]
[1 1]
sage: G = GO(4, QQ)
sage: G.invariant_bilinear_form()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: GO3m = GO(3, QQ, invariant_form=(1,0,0, 0,2,0, 0,0,3))
sage: GO3m.invariant_bilinear_form()
[1 0 0]
[0 2 0]
[0 0 3]
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> GO(Integer(2),Integer(3),+Integer(1)).invariant_bilinear_form()
[0 1]
[1 0]
>>> GO(Integer(2),Integer(3),-Integer(1)).invariant_bilinear_form()
[2 1]
[1 1]
>>> G = GO(Integer(4), QQ)
>>> G.invariant_bilinear_form()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
>>> GO3m = GO(Integer(3), QQ, invariant_form=(Integer(1),Integer(0),
↪Integer(0), Integer(0),Integer(2),Integer(0), Integer(0),Integer(0),
↪Integer(3)))
>>> GO3m.invariant_bilinear_form()
[1 0 0]
[0 2 0]
[0 0 3]
```

`sage.groups.matrix_gps.orthogonal.`**SO**(*n*, *R*, *e=None*, *var='a'*, *invariant_form=None*)

Return the special orthogonal group.

The special orthogonal group $GO(n, R)$ consists of all $n \times n$ matrices with determinant one over the ring R preserving an n-ary positive definite quadratic form. In cases where there are multiple non-isomorphic quadratic forms, additional data needs to be specified to disambiguate.

Note

This group is also available via `groups.matrix.SO()`.

INPUT:

- `n` – integer; the degree
- `R` – ring or an integer; if an integer is specified, the corresponding finite field is used
- `e` – `+1` or `-1`, and ignored by default; only relevant for finite fields and if the degree is even: a parameter that distinguishes inequivalent invariant forms
- `var` – (default: `'a'`) variable used to represent generator of the finite field, if needed
- `invariant_form` – (optional) instances being accepted by the matrix-constructor which define a $n \times n$ square matrix over `R` describing the symmetric form to be kept invariant by the orthogonal group; the form is checked to be non-degenerate and symmetric but not to be positive definite

OUTPUT: the special orthogonal group of given degree, base ring, and choice of invariant form

EXAMPLES:

```
sage: G = SO(3,GF(5)); G
Special Orthogonal Group of degree 3 over Finite Field of size 5

sage: # needs sage.libs.gap
sage: G = SO(3,GF(5))
sage: G.gens()
(
[2 0 0]  [3 2 3]  [1 4 4]
[0 3 0]  [0 2 0]  [4 0 0]
[0 0 1], [0 3 1], [2 0 4]
)
sage: G.as_matrix_group()
Matrix group over Finite Field of size 5 with 3 generators (
[2 0 0]  [3 2 3]  [1 4 4]
[0 3 0]  [0 2 0]  [4 0 0]
[0 0 1], [0 3 1], [2 0 4]
)
```

```
>>> from sage.all import *
>>> G = SO(Integer(3),GF(Integer(5))); G
Special Orthogonal Group of degree 3 over Finite Field of size 5

>>> # needs sage.libs.gap
>>> G = SO(Integer(3),GF(Integer(5)))
>>> G.gens()
(
[2 0 0]  [3 2 3]  [1 4 4]
[0 3 0]  [0 2 0]  [4 0 0]
[0 0 1], [0 3 1], [2 0 4]
)
>>> G.as_matrix_group()
Matrix group over Finite Field of size 5 with 3 generators (
[2 0 0]  [3 2 3]  [1 4 4]
```

(continues on next page)

(continued from previous page)

```
[0 3 0]   [0 2 0]   [4 0 0]
[0 0 1], [0 3 1], [2 0 4]
)
```

Using the `invariant_form` option:

```
sage: # needs sage.rings.number_field
sage: CF3 = CyclotomicField(3); e3 = CF3.gen()
sage: m = matrix(CF3, 3, 3, [[1,e3,0], [e3,2,0], [0,0,1]])
sage: SO3  = SO(3, CF3)
sage: SO3m = SO(3, CF3, invariant_form=m)
sage: SO3 == SO3m
False
sage: SO3.invariant_form()
[1 0 0]
[0 1 0]
[0 0 1]
sage: SO3m.invariant_form()
[    1 zeta3     0]
[zeta3     2     0]
[    0     0     1]
sage: pm = Permutation([2,3,1]).to_matrix()
sage: g = SO3(pm); g in SO3; g
True
[0 0 1]
[1 0 0]
[0 1 0]
sage: SO3m(pm)
Traceback (most recent call last):
...
TypeError: matrix must be orthogonal with respect to the symmetric form
[    1 zeta3     0]
[zeta3     2     0]
[    0     0     1]

sage: SO(3, 5, invariant_form=[[1,0,0], [0,2,0], [0,0,3]])
Traceback (most recent call last):
...
NotImplementedError: invariant_form for finite groups is fixed by GAP
sage: 5+5
10
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> CF3 = CyclotomicField(Integer(3)); e3 = CF3.gen()
>>> m = matrix(CF3, Integer(3), Integer(3), [[Integer(1),e3,Integer(0)], [e3,
↪Integer(2),Integer(0)], [Integer(0),Integer(0),Integer(1)]])
>>> SO3  = SO(Integer(3), CF3)
>>> SO3m = SO(Integer(3), CF3, invariant_form=m)
>>> SO3 == SO3m
False
>>> SO3.invariant_form()
```

(continues on next page)

(continued from previous page)

```
[1 0 0]
[0 1 0]
[0 0 1]
>>> SO3m.invariant_form()
[    1 zeta3     0]
[zeta3     2     0]
[    0     0     1]
>>> pm = Permutation([Integer(2),Integer(3),Integer(1)]).to_matrix()
>>> g = SO3(pm); g in SO3; g
True
[0 0 1]
[1 0 0]
[0 1 0]
>>> SO3m(pm)
Traceback (most recent call last):
...
TypeError: matrix must be orthogonal with respect to the symmetric form
[    1 zeta3     0]
[zeta3     2     0]
[    0     0     1]

>>> SO(Integer(3), Integer(5), invariant_form=[[Integer(1),Integer(0),Integer(0)],
↪ [Integer(0),Integer(2),Integer(0)], [Integer(0),Integer(0),Integer(3)]])
Traceback (most recent call last):
...
NotImplementedError: invariant_form for finite groups is fixed by GAP
>>> Integer(5)+Integer(5)
10
```

sage.groups.matrix_gps.orthogonal.**normalize_args_e**(*degree*, *ring*, *e*)

Normalize the arguments that relate the choice of quadratic form for special orthogonal groups over finite fields.

INPUT:

- `degree` – integer; the degree of the affine group, that is, the dimension of the affine space the group is acting on
- `ring` – a ring; the base ring of the affine space
- `e` – integer; one of $+1$, 0, -1. Only relevant for finite fields and if the degree is even. A parameter that distinguishes inequivalent invariant forms.

OUTPUT: the integer `e` with values required by GAP

28.13 Orthogonal Linear Groups with GAP

class sage.groups.matrix_gps.orthogonal_gap.**OrthogonalMatrixGroup_gap**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *gap_command_string*, *category=None*)

Bases: *OrthogonalMatrixGroup_generic*, *NamedMatrixGroup_gap*, *FinitelyGeneratedMatrixGroup_gap*

The general or special orthogonal group in GAP.

invariant_bilinear_form()

Return the symmetric bilinear form preserved by the orthogonal group.

OUTPUT:

A matrix M such that, for every group element g, the identity $gmg^T = m$ holds. In characteristic different from two, this uniquely determines the orthogonal group.

EXAMPLES:

```
sage: G = GO(4, GF(7), -1)
sage: G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 2 0]
[0 0 0 2]

sage: G = GO(4, GF(7), +1)
sage: G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 6 0]
[0 0 0 2]

sage: G = SO(4, GF(7), -1)
sage: G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 2 0]
[0 0 0 2]
```

```
>>> from sage.all import *
>>> G = GO(Integer(4), GF(Integer(7)), -Integer(1))
>>> G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 2 0]
[0 0 0 2]

>>> G = GO(Integer(4), GF(Integer(7)), +Integer(1))
>>> G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 6 0]
[0 0 0 2]

>>> G = SO(Integer(4), GF(Integer(7)), -Integer(1))
>>> G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 2 0]
[0 0 0 2]
```

invariant_form()

Return the symmetric bilinear form preserved by the orthogonal group.

OUTPUT:

A matrix M such that, for every group element g, the identity $gmg^T = m$ holds. In characteristic different from two, this uniquely determines the orthogonal group.

EXAMPLES:

```
sage: G = GO(4, GF(7), -1)
sage: G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 2 0]
[0 0 0 2]

sage: G = GO(4, GF(7), +1)
sage: G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 6 0]
[0 0 0 2]

sage: G = SO(4, GF(7), -1)
sage: G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 2 0]
[0 0 0 2]
```

```
>>> from sage.all import *
>>> G = GO(Integer(4), GF(Integer(7)), -Integer(1))
>>> G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 2 0]
[0 0 0 2]

>>> G = GO(Integer(4), GF(Integer(7)), +Integer(1))
>>> G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 6 0]
[0 0 0 2]

>>> G = SO(Integer(4), GF(Integer(7)), -Integer(1))
>>> G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 2 0]
[0 0 0 2]
```

invariant_quadratic_form()

Return the quadratic form preserved by the orthogonal group.

OUTPUT:

The matrix Q defining "orthogonal" as follows. The matrix determines a quadratic form q on the natural vector space V, on which G acts, by $q(v) = vQv^t$. A matrix M is an element of the orthogonal group if $q(v) = q(vM)$ for all $v \in V$.

EXAMPLES:

```
sage: G = GO(4, GF(7), -1)
sage: G.invariant_quadratic_form()
[0 1 0 0]
[0 0 0 0]
[0 0 1 0]
[0 0 0 1]

sage: G = GO(4, GF(7), +1)
sage: G.invariant_quadratic_form()
[0 1 0 0]
[0 0 0 0]
[0 0 3 0]
[0 0 0 1]

sage: G = GO(4, QQ)
sage: G.invariant_quadratic_form()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

sage: G = SO(4, GF(7), -1)
sage: G.invariant_quadratic_form()
[0 1 0 0]
[0 0 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> G = GO(Integer(4), GF(Integer(7)), -Integer(1))
>>> G.invariant_quadratic_form()
[0 1 0 0]
[0 0 0 0]
[0 0 1 0]
[0 0 0 1]

>>> G = GO(Integer(4), GF(Integer(7)), +Integer(1))
>>> G.invariant_quadratic_form()
[0 1 0 0]
[0 0 0 0]
[0 0 3 0]
[0 0 0 1]

>>> G = GO(Integer(4), QQ)
>>> G.invariant_quadratic_form()
[1 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

>>> G = SO(Integer(4), GF(Integer(7)), -Integer(1))
>>> G.invariant_quadratic_form()
[0 1 0 0]
[0 0 0 0]
[0 0 1 0]
[0 0 0 1]
```

28.14 Groups of isometries

Let $M = \mathbf{Z}^n$ or $\mathbf{Q}^n$, $b : M \times M \to \mathbf{Q}$ a bilinear form and $f : M \to M$ a linear map. We say that f is an isometry if for all elements x, y of M we have that $b(x, y) = b(f(x), f(y))$. A group of isometries is a subgroup of $GL(M)$ consisting of isometries.

EXAMPLES:

```
sage: L = IntegralLattice("D4")                                             #␣
↪needs sage.graphs
sage: O = L.orthogonal_group(); O                                           #␣
↪needs sage.graphs
Group of isometries with 3 generators (
[0 0 0 1]   [ 1  1  0  0]   [ 1  0  0  0]
[0 1 0 0]   [ 0  0  1  0]   [-1 -1 -1 -1]
[0 0 1 0]   [ 0  1  0  1]   [ 0  0  1  0]
[1 0 0 0], [ 0 -1 -1  0], [ 0  0  0  1]
)
```

```
>>> from sage.all import *
>>> L = IntegralLattice("D4")                                               #␣
↪needs sage.graphs
>>> O = L.orthogonal_group(); O                                             #␣
↪needs sage.graphs
Group of isometries with 3 generators (
[0 0 0 1]   [ 1  1  0  0]   [ 1  0  0  0]
[0 1 0 0]   [ 0  0  1  0]   [-1 -1 -1 -1]
[0 0 1 0]   [ 0  1  0  1]   [ 0  0  1  0]
[1 0 0 0], [ 0 -1 -1  0], [ 0  0  0  1]
)
```

Basic functionality is provided by GAP:

```
sage: O.cardinality()                                                       #␣
↪needs sage.graphs
1152
sage: len(O.conjugacy_classes_representatives())                            #␣
↪needs sage.graphs
25
```

```
>>> from sage.all import *
>>> O.cardinality()                                                  #␣
↪needs sage.graphs
1152
>>> len(O.conjugacy_classes_representatives())                       #␣
↪needs sage.graphs
25
```

AUTHORS:

- Simon Brandhorst (2018-02): First created

class sage.groups.matrix_gps.isometries.**GroupActionOnQuotientModule**(*MatrixGroup*, *quotient_module*, *is_left=False*)

Bases: `Action`

Matrix group action on a quotient module from the right.

INPUT:

- `MatrixGroup` – the group acting `GroupOfIsometries`
- `submodule` – an invariant quotient module
- `is_left` – boolean (default: `False`)

EXAMPLES:

```
sage: from sage.groups.matrix_gps.isometries import GroupOfIsometries
sage: S = span(ZZ,[[0,1]])
sage: Q = S/(6*S)
sage: g = Matrix(QQ,2,[1,0,0,-1])
sage: G = GroupOfIsometries(2, ZZ, [g], invariant_bilinear_form=matrix.
↪identity(2), invariant_quotient_module=Q)
sage: g = G.an_element()
sage: x = Q.an_element()
sage: x*g
(5)
sage: (x*g).parent()
Finitely generated module V/W over Integer Ring with invariants (6)
```

```
>>> from sage.all import *
>>> from sage.groups.matrix_gps.isometries import GroupOfIsometries
>>> S = span(ZZ,[[Integer(0),Integer(1)]])
>>> Q = S/(Integer(6)*S)
>>> g = Matrix(QQ,Integer(2),[Integer(1),Integer(0),Integer(0),-Integer(1)])
>>> G = GroupOfIsometries(Integer(2), ZZ, [g], invariant_bilinear_form=matrix.
↪identity(Integer(2)), invariant_quotient_module=Q)
>>> g = G.an_element()
>>> x = Q.an_element()
>>> x*g
(5)
>>> (x*g).parent()
Finitely generated module V/W over Integer Ring with invariants (6)
```

class sage.groups.matrix_gps.isometries.**GroupActionOnSubmodule**(*MatrixGroup*, *submodule*, *is_left=False*)

Bases: Action

Matrix group action on a submodule from the right.

INPUT:

- MatrixGroup – an instance of *GroupOfIsometries*
- submodule – an invariant submodule
- is_left – boolean (default: False)

EXAMPLES:

```
sage: from sage.groups.matrix_gps.isometries import GroupOfIsometries
sage: S = span(ZZ, [[0,1]])
sage: g = Matrix(QQ, 2, [1,0,0,-1])
sage: G = GroupOfIsometries(2, ZZ, [g],
....:                         invariant_bilinear_form=matrix.identity(2),
....:                         invariant_submodule=S)
sage: g = G.an_element()
sage: x = S.an_element()
sage: x*g
(0, -1)
sage: (x*g).parent()
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1]
```

```
>>> from sage.all import *
>>> from sage.groups.matrix_gps.isometries import GroupOfIsometries
>>> S = span(ZZ, [[Integer(0),Integer(1)]])
>>> g = Matrix(QQ, Integer(2), [Integer(1),Integer(0),Integer(0),-Integer(1)])
>>> G = GroupOfIsometries(Integer(2), ZZ, [g],
...                         invariant_bilinear_form=matrix.identity(Integer(2)),
...                         invariant_submodule=S)
>>> g = G.an_element()
>>> x = S.an_element()
>>> x*g
(0, -1)
>>> (x*g).parent()
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1]
```

class sage.groups.matrix_gps.isometries.**GroupOfIsometries**(*degree*, *base_ring*, *gens*, *invariant_bilinear_form*, *category=None*, *check=True*, *invariant_submodule=None*, *invariant_quotient_module=None*)

Bases: *FinitelyGeneratedMatrixGroup_gap*

A base class for Orthogonal matrix groups with a gap backend.

Main difference to `OrthogonalMatrixGroup_gap` is that we can specify generators and a bilinear form. Following GAP, the group action is from the right.

INPUT:

- `degree` – integer; the degree (matrix size) of the matrix
- `base_ring` – ring, the base ring of the matrices
- `gens` – list of matrices over the base ring
- `invariant_bilinear_form` – a symmetric matrix
- `category` – (default: `None`) a category of groups
- `check` – boolean (default: `True`); check if the generators preserve the bilinear form
- `invariant_submodule` – a submodule preserved by the group action (default: `None`); registers an action on this submodule
- `invariant_quotient_module` – a quotient module preserved by the group action (default: `None`) registers an action on this quotient module

EXAMPLES:

```
sage: from sage.groups.matrix_gps.isometries import GroupOfIsometries
sage: bil = Matrix(ZZ, 2, [3,2,2,3])
sage: gens = [-Matrix(ZZ, 2, [0,1,1,0])]
sage: O = GroupOfIsometries(2, ZZ, gens, bil)
sage: O
Group of isometries with 1 generator (
[ 0 -1]
[-1  0]
)
sage: O.order()
2
```

```
>>> from sage.all import *
>>> from sage.groups.matrix_gps.isometries import GroupOfIsometries
>>> bil = Matrix(ZZ, Integer(2), [Integer(3),Integer(2),Integer(2),Integer(3)])
>>> gens = [-Matrix(ZZ, Integer(2), [Integer(0),Integer(1),Integer(1),
↪Integer(0)])]
>>> O = GroupOfIsometries(Integer(2), ZZ, gens, bil)
>>> O
Group of isometries with 1 generator (
[ 0 -1]
[-1  0]
)
>>> O.order()
2
```

Infinite groups are O.K. too:

```
sage: bil = Matrix(ZZ,4,[0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0])
sage: f = Matrix(ZZ,4,[0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, -1, 1, 1, 1])
sage: O = GroupOfIsometries(2, ZZ, [f], bil)
sage: O.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> bil = Matrix(ZZ,Integer(4),[Integer(0), Integer(1), Integer(1), Integer(1),␣
↪Integer(1), Integer(0), Integer(1), Integer(1), Integer(1), Integer(1),␣
↪Integer(0), Integer(1), Integer(1), Integer(1), Integer(1), Integer(0)])
>>> f = Matrix(ZZ,Integer(4),[Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(1), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), -Integer(1), Integer(1), Integer(1), Integer(1)])
>>> O = GroupOfIsometries(Integer(2), ZZ, [f], bil)
>>> O.cardinality()
+Infinity
```

invariant_bilinear_form()

Return the symmetric bilinear form preserved by the orthogonal group.

OUTPUT: the matrix defining the bilinear form

EXAMPLES:

```
sage: from sage.groups.matrix_gps.isometries import GroupOfIsometries
sage: bil = Matrix(ZZ,2,[3,2,2,3])
sage: gens = [-Matrix(ZZ,2,[0,1,1,0])]
sage: O = GroupOfIsometries(2,ZZ,gens,bil)
sage: O.invariant_bilinear_form()
[3 2]
[2 3]
```

```
>>> from sage.all import *
>>> from sage.groups.matrix_gps.isometries import GroupOfIsometries
>>> bil = Matrix(ZZ,Integer(2),[Integer(3),Integer(2),Integer(2),Integer(3)])
>>> gens = [-Matrix(ZZ,Integer(2),[Integer(0),Integer(1),Integer(1),
↪Integer(0)])]
>>> O = GroupOfIsometries(Integer(2),ZZ,gens,bil)
>>> O.invariant_bilinear_form()
[3 2]
[2 3]
```

28.15 Symplectic Linear Groups

EXAMPLES:

```
sage: G = Sp(4, GF(7));  G
Symplectic Group of degree 4 over Finite Field of size 7

sage: # needs sage.libs.gap
sage: g = prod(G.gens());  g
[3 0 3 0]
[1 0 0 0]
[0 1 0 1]
[0 2 0 0]
sage: m = g.matrix()
sage: m * G.invariant_form() * m.transpose() == G.invariant_form()
True
```

(continues on next page)

(continued from previous page)

```
sage: G.order()
276595200
```

```
>>> from sage.all import *
>>> G = Sp(Integer(4), GF(Integer(7)));  G
Symplectic Group of degree 4 over Finite Field of size 7

>>> # needs sage.libs.gap
>>> g = prod(G.gens());  g
[3 0 3 0]
[1 0 0 0]
[0 1 0 1]
[0 2 0 0]
>>> m = g.matrix()
>>> m * G.invariant_form() * m.transpose() == G.invariant_form()
True
>>> G.order()
276595200
```

AUTHORS:

- David Joyner (2006-03): initial version, modified from special_linear (by W. Stein)
- Volker Braun (2013-1) port to new Parent, libGAP, extreme refactoring.
- Sebastian Oehms (2018-8) add option for user defined invariant bilinear form and bug-fix in `invariant_form()` (see Issue #26028)

`sage.groups.matrix_gps.symplectic.`**`Sp`**`(`*n*, *R*, *var='a'*, *invariant_form=None*`)`

Return the symplectic group.

The special linear group $GL(d, R)$ consists of all $d \times d$ matrices that are invertible over the ring R with determinant one.

Note

This group is also available via `groups.matrix.Sp()`.

INPUT:

- `n` – positive integer
- `R` – ring or an integer; if an integer is specified, the corresponding finite field is used
- `var` – (default: `'a'`) variable used to represent generator of the finite field, if needed
- `invariant_form` – (optional) instances being accepted by the matrix-constructor which define a $n \times n$ square matrix over `R` describing the alternating form to be kept invariant by the symplectic group

EXAMPLES:

```
sage: Sp(4, 5)
Symplectic Group of degree 4 over Finite Field of size 5

sage: Sp(4, IntegerModRing(15))
Symplectic Group of degree 4 over Ring of integers modulo 15
```

(continues on next page)

(continued from previous page)

```
sage: Sp(3, GF(7))
Traceback (most recent call last):
...
ValueError: the degree must be even
```

```
>>> from sage.all import *
>>> Sp(Integer(4), Integer(5))
Symplectic Group of degree 4 over Finite Field of size 5

>>> Sp(Integer(4), IntegerModRing(Integer(15)))
Symplectic Group of degree 4 over Ring of integers modulo 15

>>> Sp(Integer(3), GF(Integer(7)))
Traceback (most recent call last):
...
ValueError: the degree must be even
```

Using the `invariant_form` option:

```
sage: m = matrix(QQ, 4,4, [[0, 0, 1, 0], [0, 0, 0, 2], [-1, 0, 0, 0], [0, -2, 0,␣
↪0]])
sage: Sp4m = Sp(4, QQ, invariant_form=m)
sage: Sp4 = Sp(4, QQ)
sage: Sp4 == Sp4m
False
sage: Sp4.invariant_form()
[ 0  0  0  1]
[ 0  0  1  0]
[ 0 -1  0  0]
[-1  0  0  0]
sage: Sp4m.invariant_form()
[ 0  0  1  0]
[ 0  0  0  2]
[-1  0  0  0]
[ 0 -2  0  0]
sage: pm = Permutation([2,1,4,3]).to_matrix()
sage: g = Sp4(pm); g in Sp4; g
True
[0 1 0 0]
[1 0 0 0]
[0 0 0 1]
[0 0 1 0]
sage: Sp4m(pm)
Traceback (most recent call last):
...
TypeError: matrix must be symplectic with respect to the alternating form
[ 0  0  1  0]
[ 0  0  0  2]
[-1  0  0  0]
[ 0 -2  0  0]
```

(continues on next page)

(continued from previous page)

```
sage: Sp(4,3, invariant_form=[[0,0,0,1],[0,0,1,0],[0,2,0,0], [2,0,0,0]])
Traceback (most recent call last):
...
NotImplementedError: invariant_form for finite groups is fixed by GAP
```

```
>>> from sage.all import *
>>> m = matrix(QQ, Integer(4),Integer(4), [[Integer(0), Integer(0), Integer(1),
↪Integer(0)], [Integer(0), Integer(0), Integer(0), Integer(2)], [-Integer(1),
↪Integer(0), Integer(0), Integer(0)], [Integer(0), -Integer(2), Integer(0),
↪Integer(0)]])
>>> Sp4m = Sp(Integer(4), QQ, invariant_form=m)
>>> Sp4 = Sp(Integer(4), QQ)
>>> Sp4 == Sp4m
False
>>> Sp4.invariant_form()
[ 0  0  0  1]
[ 0  0  1  0]
[ 0 -1  0  0]
[-1  0  0  0]
>>> Sp4m.invariant_form()
[ 0  0  1  0]
[ 0  0  0  2]
[-1  0  0  0]
[ 0 -2  0  0]
>>> pm = Permutation([Integer(2),Integer(1),Integer(4),Integer(3)]).to_matrix()
>>> g = Sp4(pm); g in Sp4; g
True
[0 1 0 0]
[1 0 0 0]
[0 0 0 1]
[0 0 1 0]
>>> Sp4m(pm)
Traceback (most recent call last):
...
TypeError: matrix must be symplectic with respect to the alternating form
[ 0  0  1  0]
[ 0  0  0  2]
[-1  0  0  0]
[ 0 -2  0  0]

>>> Sp(Integer(4),Integer(3), invariant_form=[[Integer(0),Integer(0),Integer(0),
↪Integer(1)],[Integer(0),Integer(0),Integer(1),Integer(0)],[Integer(0),
↪Integer(2),Integer(0),Integer(0)], [Integer(2),Integer(0),Integer(0),
↪Integer(0)]])
Traceback (most recent call last):
...
NotImplementedError: invariant_form for finite groups is fixed by GAP
```

class `sage.groups.matrix_gps.symplectic.`**`SymplecticMatrixGroup_generic`**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *category=None*, *invariant_form=None*)

Bases: *NamedMatrixGroup_generic*

Symplectic Group over arbitrary rings.

EXAMPLES:

```
sage: Sp43 = Sp(4,3); Sp43
Symplectic Group of degree 4 over Finite Field of size 3
sage: latex(Sp43)
\text{Sp}_{4}(\Bold{F}_{3})

sage: Sp4m = Sp(4, QQ, invariant_form=(0, 0, 1, 0,  0, 0, 0, 2,
....:                                  -1, 0, 0, 0, 0, -2, 0, 0)); Sp4m
Symplectic Group of degree 4 over Rational Field
 with respect to alternating bilinear form
[ 0  0  1  0]
[ 0  0  0  2]
[-1  0  0  0]
[ 0 -2  0  0]
sage: latex(Sp4m)
\text{Sp}_{4}(\Bold{Q})\text{ with respect to alternating bilinear form}\left(\
↪begin{array}{rrrr}
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 2 \\
-1 & 0 & 0 & 0 \\
0 & -2 & 0 & 0
\end{array}\right)
```

```
>>> from sage.all import *
>>> Sp43 = Sp(Integer(4),Integer(3)); Sp43
Symplectic Group of degree 4 over Finite Field of size 3
>>> latex(Sp43)
\text{Sp}_{4}(\Bold{F}_{3})

>>> Sp4m = Sp(Integer(4), QQ, invariant_form=(Integer(0), Integer(0), Integer(1),␣
↪Integer(0),  Integer(0), Integer(0), Integer(0), Integer(2),
...                                    -Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), -Integer(2), Integer(0), Integer(0))); Sp4m
Symplectic Group of degree 4 over Rational Field
 with respect to alternating bilinear form
[ 0  0  1  0]
[ 0  0  0  2]
[-1  0  0  0]
[ 0 -2  0  0]
>>> latex(Sp4m)
\text{Sp}_{4}(\Bold{Q})\text{ with respect to alternating bilinear form}\left(\
↪begin{array}{rrrr}
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 2 \\
-1 & 0 & 0 & 0 \\
0 & -2 & 0 & 0
\end{array}\right)
```

invariant_form()

Return the quadratic form preserved by the symplectic group.

OUTPUT: a matrix

EXAMPLES:

```
sage: Sp(4, QQ).invariant_form()
[ 0  0  0  1]
[ 0  0  1  0]
[ 0 -1  0  0]
[-1  0  0  0]
```

```
>>> from sage.all import *
>>> Sp(Integer(4), QQ).invariant_form()
[ 0  0  0  1]
[ 0  0  1  0]
[ 0 -1  0  0]
[-1  0  0  0]
```

28.16 Symplectic Linear Groups with GAP

class sage.groups.matrix_gps.symplectic_gap.**SymplecticMatrixGroup_gap**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *gap_command_string*, *category=None*)

Bases: *SymplecticMatrixGroup_generic*, *NamedMatrixGroup_gap*, *FinitelyGeneratedMatrixGroup_gap*

Symplectic group in GAP.

EXAMPLES:

```
sage: Sp(2,4)                                                                   #␣
↪needs sage.rings.finite_rings
Symplectic Group of degree 2 over Finite Field in a of size 2^2

sage: latex(Sp(4,5))
\text{Sp}_{4}(\Bold{F}_{5})
```

```
>>> from sage.all import *
>>> Sp(Integer(2),Integer(4))                                                   ␣
↪               # needs sage.rings.finite_rings
Symplectic Group of degree 2 over Finite Field in a of size 2^2

>>> latex(Sp(Integer(4),Integer(5)))
\text{Sp}_{4}(\Bold{F}_{5})
```

invariant_form()

Return the quadratic form preserved by the symplectic group.

OUTPUT: a matrix

EXAMPLES:

```
sage: Sp(4, GF(3)).invariant_form()
[0 0 0 1]
[0 0 1 0]
[0 2 0 0]
[2 0 0 0]
```

```
>>> from sage.all import *
>>> Sp(Integer(4), GF(Integer(3))).invariant_form()
[0 0 0 1]
[0 0 1 0]
[0 2 0 0]
[2 0 0 0]
```

28.17 Unitary Groups $GU(n, q)$ and $SU(n, q)$

These are $n \times n$ unitary matrices with entries in $GF(q^2)$.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: G = SU(3,5)
sage: G.order()                                                          #␣
↪needs sage.libs.gap
378000
sage: G
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
sage: G.gens()                                                           #␣
↪needs sage.libs.gap
(
[      a       0       0]  [4*a    4    1]
[      0 2*a + 2       0]  [  4    4    0]
[      0       0     3*a], [  1    0    0]
)
sage: G.base_ring()
Finite Field in a of size 5^2
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> G = SU(Integer(3),Integer(5))
>>> G.order()                                                          #␣
↪needs sage.libs.gap
378000
>>> G
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
>>> G.gens()                                                           #␣
↪needs sage.libs.gap
(
[      a       0       0]  [4*a    4    1]
[      0 2*a + 2       0]  [  4    4    0]
[      0       0     3*a], [  1    0    0]
)
```

(continues on next page)

(continued from previous page)

```
>>> G.base_ring()
Finite Field in a of size 5^2
```

AUTHORS:

- David Joyner (2006-03): initial version, modified from special_linear (by W. Stein)
- David Joyner (2006-05): minor additions (examples, _latex_, __str__, gens)
- William Stein (2006-12): rewrite
- Volker Braun (2013-1) port to new Parent, libGAP, extreme refactoring.
- Sebastian Oehms (2018-8) add `_UG`, `invariant_form()`, option for user defined invariant bilinear form, and bug-fix in `_check_matrix` (see Issue #26028)

`sage.groups.matrix_gps.unitary.`**`GU`**`(`*n*, *R*, *var='a'*, *invariant_form=None*`)`

Return the general unitary group.

The general unitary group $GU(d, R)$ consists of all $d \times d$ matrices that preserve a nondegenerate sesquilinear form over the ring R.

Note: For a finite field, the matrices that preserve a sesquilinear form over $\mathbf{F}_q$ live over $\mathbf{F}_{q^2}$. So `GU(n,q)` for a prime power q constructs the matrix group over the base ring `GF(q^2)`.

Note: This group is also available via `groups.matrix.GU()`.

INPUT:

- `n` – positive integer
- `R` – ring or an integer; if an integer is specified, the corresponding finite field is used
- `var` – (default: `'a'`) variable used to represent generator of the finite field, if needed
- `invariant_form` – (optional) instances being accepted by the matrix-constructor which define a $n \times n$ square matrix over R describing the hermitian form to be kept invariant by the unitary group; the form is checked to be non-degenerate and hermitian but not to be positive definite

OUTPUT: the general unitary group

EXAMPLES:

```
sage: G = GU(3, 7); G                                                    #␣
↪needs sage.rings.finite_rings
General Unitary Group of degree 3 over Finite Field in a of size 7^2
sage: G.gens()                                                           #␣
↪needs sage.libs.gap sage.rings.finite_rings
(
[  a   0   0]  [6*a   6   1]
[  0   1   0]  [  6   6   0]
[  0   0 5*a], [  1   0   0]
```

(continues on next page)

(continued from previous page)

```
)
sage: GU(2, QQ)
General Unitary Group of degree 2 over Rational Field

sage: G = GU(3, 5, var='beta')                                               #␣
↪needs sage.rings.finite_rings
sage: G.base_ring()                                                          #␣
↪needs sage.rings.finite_rings
Finite Field in beta of size 5^2
sage: G.gens()                                                               #␣
↪needs sage.libs.gap sage.rings.finite_rings
(
[  beta      0      0]  [4*beta      4      1]
[     0      1      0]  [     4      4      0]
[     0      0 3*beta], [     1      0      0]
)
```

```
>>> from sage.all import *
>>> G = GU(Integer(3), Integer(7)); G                                        ␣
↪              # needs sage.rings.finite_rings
General Unitary Group of degree 3 over Finite Field in a of size 7^2
>>> G.gens()                                                               #␣
↪needs sage.libs.gap sage.rings.finite_rings
(
[  a   0   0]  [6*a   6   1]
[  0   1   0]  [  6   6   0]
[  0   0 5*a], [  1   0   0]
)
>>> GU(Integer(2), QQ)
General Unitary Group of degree 2 over Rational Field

>>> G = GU(Integer(3), Integer(5), var='beta')                               ␣
↪              # needs sage.rings.finite_rings
>>> G.base_ring()                                                          #␣
↪needs sage.rings.finite_rings
Finite Field in beta of size 5^2
>>> G.gens()                                                               #␣
↪needs sage.libs.gap sage.rings.finite_rings
(
[  beta      0      0]  [4*beta      4      1]
[     0      1      0]  [     4      4      0]
[     0      0 3*beta], [     1      0      0]
)
```

Using the `invariant_form` option:

```
sage: # needs sage.libs.gap sage.rings.number_field
sage: UCF = UniversalCyclotomicField(); e5 = UCF.gen(5)
sage: m = matrix(UCF, 3, 3, [[1,e5,0], [e5.conjugate(),2,0], [0,0,1]])
sage: G  = GU(3, UCF)
sage: Gm = GU(3, UCF, invariant_form=m)
sage: G == Gm
```

(continues on next page)

(continued from previous page)

```
False
sage: G.invariant_form()
[1 0 0]
[0 1 0]
[0 0 1]
sage: Gm.invariant_form()
[     1   E(5)      0]
[E(5)^4      2      0]
[     0      0      1]
sage: pm = Permutation((1,2,3)).to_matrix()
sage: g = G(pm); g in G; g                                                    #␣
↪needs sage.combinat
True
[0 0 1]
[1 0 0]
[0 1 0]
sage: Gm(pm)                                                                  #␣
↪needs sage.combinat
Traceback (most recent call last):
...
TypeError: matrix must be unitary with respect to the hermitian form
[     1   E(5)      0]
[E(5)^4      2      0]
[     0      0      1]

sage: GU(3, 3, invariant_form=[[1,0,0], [0,2,0], [0,0,1]])                    #␣
↪needs sage.libs.pari
Traceback (most recent call last):
...
NotImplementedError: invariant_form for finite groups is fixed by GAP

sage: GU(2, QQ, invariant_form=[[1,0], [2,0]])
Traceback (most recent call last):
...
ValueError: invariant_form must be non-degenerate
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap sage.rings.number_field
>>> UCF = UniversalCyclotomicField(); e5 = UCF.gen(Integer(5))
>>> m = matrix(UCF, Integer(3), Integer(3), [[Integer(1),e5,Integer(0)], [e5.
↪conjugate(),Integer(2),Integer(0)], [Integer(0),Integer(0),Integer(1)]])
>>> G  = GU(Integer(3), UCF)
>>> Gm = GU(Integer(3), UCF, invariant_form=m)
>>> G == Gm
False
>>> G.invariant_form()
[1 0 0]
[0 1 0]
[0 0 1]
>>> Gm.invariant_form()
[     1   E(5)      0]
[E(5)^4      2      0]
```

(continues on next page)

(continued from previous page)

```
[      0       0       1]
>>> pm = Permutation((Integer(1),Integer(2),Integer(3))).to_matrix()
>>> g = G(pm); g in G; g                                                    #␣
↪needs sage.combinat
True
[0 0 1]
[1 0 0]
[0 1 0]
>>> Gm(pm)                                                                  #␣
↪needs sage.combinat
Traceback (most recent call last):
...
TypeError: matrix must be unitary with respect to the hermitian form
[      1    E(5)       0]
[E(5)^4       2       0]
[      0       0       1]

>>> GU(Integer(3), Integer(3), invariant_form=[[Integer(1),Integer(0),Integer(0)],
↪ [Integer(0),Integer(2),Integer(0)], [Integer(0),Integer(0),Integer(1)]])     ␣
↪                # needs sage.libs.pari
Traceback (most recent call last):
...
NotImplementedError: invariant_form for finite groups is fixed by GAP

>>> GU(Integer(2), QQ, invariant_form=[[Integer(1),Integer(0)], [Integer(2),
↪Integer(0)]])
Traceback (most recent call last):
...
ValueError: invariant_form must be non-degenerate
```

`sage.groups.matrix_gps.unitary.SU(n, R, var='a', invariant_form=None)`

The special unitary group $SU(d, R)$ consists of all $d \times d$ matrices that preserve a nondegenerate sesquilinear form over the ring R and have determinant 1.

Note

For a finite field the matrices that preserve a sesquilinear form over $\mathbf{F}_q$ live over $\mathbf{F}_{q^2}$. So `SU(n,q)` for a prime power q constructs the matrix group over the base ring `GF(q^2)`.

Note

This group is also available via `groups.matrix.SU()`.

INPUT:

- `n` – positive integer
- `R` – ring or an integer; if an integer is specified, the corresponding finite field is used
- `var` – (default: `'a'`) variable used to represent generator of the finite field, if needed
- `invariant_form` – (optional) instances being accepted by the matrix-constructor which define a $n \times n$

square matrix over R describing the hermitian form to be kept invariant by the unitary group; the form is checked to be non-degenerate and hermitian but not to be positive definite

OUTPUT: the special unitary group

EXAMPLES:

```
sage: SU(3,5)                                                             #␣
↪needs sage.rings.finite_rings
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
sage: SU(3, GF(5))                                                        #␣
↪needs sage.rings.finite_rings
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
sage: SU(3, QQ)
Special Unitary Group of degree 3 over Rational Field
```

```
>>> from sage.all import *
>>> SU(Integer(3),Integer(5))                                              ␣
↪             # needs sage.rings.finite_rings
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
>>> SU(Integer(3), GF(Integer(5)))                                         ␣
↪             # needs sage.rings.finite_rings
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
>>> SU(Integer(3), QQ)
Special Unitary Group of degree 3 over Rational Field
```

Using the `invariant_form` option:

```
sage: # needs sage.rings.number_field
sage: CF3 = CyclotomicField(3); e3 = CF3.gen()
sage: m = matrix(CF3, 3, 3, [[1,e3,0], [e3.conjugate(),2,0], [0,0,1]])
sage: G  = SU(3, CF3)
sage: Gm = SU(3, CF3, invariant_form=m)
sage: G == Gm
False
sage: G.invariant_form()
[1 0 0]
[0 1 0]
[0 0 1]
sage: Gm.invariant_form()
[         1      zeta3          0]
[-zeta3 - 1          2          0]
[         0          0          1]
sage: pm = Permutation((1,2,3)).to_matrix()
sage: G(pm)                                                               #␣
↪needs sage.combinat
[0 0 1]
[1 0 0]
[0 1 0]
sage: Gm(pm)                                                              #␣
↪needs sage.combinat
Traceback (most recent call last):
...
TypeError: matrix must be unitary with respect to the hermitian form
```

(continues on next page)

(continued from previous page)

```
[         1      zeta3           0]
[-zeta3 - 1          2           0]
[         0          0           1]

sage: SU(3, 5, invariant_form=[[1,0,0], [0,2,0], [0,0,3]])                    #␣
↪needs sage.rings.finite_rings
Traceback (most recent call last):
...
NotImplementedError: invariant_form for finite groups is fixed by GAP
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> CF3 = CyclotomicField(Integer(3)); e3 = CF3.gen()
>>> m = matrix(CF3, Integer(3), Integer(3), [[Integer(1),e3,Integer(0)], [e3.
↪conjugate(),Integer(2),Integer(0)], [Integer(0),Integer(0),Integer(1)]])
>>> G  = SU(Integer(3), CF3)
>>> Gm = SU(Integer(3), CF3, invariant_form=m)
>>> G == Gm
False
>>> G.invariant_form()
[1 0 0]
[0 1 0]
[0 0 1]
>>> Gm.invariant_form()
[         1      zeta3           0]
[-zeta3 - 1          2           0]
[         0          0           1]
>>> pm = Permutation((Integer(1),Integer(2),Integer(3))).to_matrix()
>>> G(pm)                                                                     #␣
↪needs sage.combinat
[0 0 1]
[1 0 0]
[0 1 0]
>>> Gm(pm)                                                                    #␣
↪needs sage.combinat
Traceback (most recent call last):
...
TypeError: matrix must be unitary with respect to the hermitian form
[         1      zeta3           0]
[-zeta3 - 1          2           0]
[         0          0           1]

>>> SU(Integer(3), Integer(5), invariant_form=[[Integer(1),Integer(0),Integer(0)],
↪ [Integer(0),Integer(2),Integer(0)], [Integer(0),Integer(0),Integer(3)]])      ␣
↪                # needs sage.rings.finite_rings
Traceback (most recent call last):
...
NotImplementedError: invariant_form for finite groups is fixed by GAP
```

class sage.groups.matrix_gps.unitary.**UnitaryMatrixGroup_generic**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *category=None*, *invariant_form=None*)

Bases: *NamedMatrixGroup_generic*

General Unitary Group over arbitrary rings.

EXAMPLES:

```
sage: G = GU(3, GF(7)); G                                                        #␣
↪needs sage.rings.finite_rings
General Unitary Group of degree 3 over Finite Field in a of size 7^2
sage: latex(G)                                                                   #␣
↪needs sage.rings.finite_rings
\text{GU}_{3}(\Bold{F}_{7^{2}})

sage: G = SU(3, GF(5));  G                                                       #␣
↪needs sage.rings.finite_rings
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
sage: latex(G)                                                                   #␣
↪needs sage.rings.finite_rings
\text{SU}_{3}(\Bold{F}_{5^{2}})

sage: # needs sage.rings.number_field
sage: CF3 = CyclotomicField(3); e3 = CF3.gen()
sage: m = matrix(CF3, 3, 3, [[1,e3,0], [e3.conjugate(),2,0], [0,0,1]])
sage: G = SU(3, CF3, invariant_form=m)
sage: latex(G)
\text{SU}_{3}(\Bold{Q}(\zeta_{3}))\text{ with respect to positive definite␣
↪hermitian form }\left(\begin{array}{rrr}
1 & \zeta_{3} & 0 \\
-\zeta_{3} - 1 & 2 & 0 \\
0 & 0 & 1
\end{array}\right)
```

```
>>> from sage.all import *
>>> G = GU(Integer(3), GF(Integer(7))); G                                        ␣
↪                # needs sage.rings.finite_rings
General Unitary Group of degree 3 over Finite Field in a of size 7^2
>>> latex(G)                                                                   #␣
↪needs sage.rings.finite_rings
\text{GU}_{3}(\Bold{F}_{7^{2}})

>>> G = SU(Integer(3), GF(Integer(5)));  G                                       ␣
↪                # needs sage.rings.finite_rings
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
>>> latex(G)                                                                   #␣
↪needs sage.rings.finite_rings
\text{SU}_{3}(\Bold{F}_{5^{2}})

>>> # needs sage.rings.number_field
>>> CF3 = CyclotomicField(Integer(3)); e3 = CF3.gen()
>>> m = matrix(CF3, Integer(3), Integer(3), [[Integer(1),e3,Integer(0)], [e3.
↪conjugate(),Integer(2),Integer(0)], [Integer(0),Integer(0),Integer(1)]])
>>> G = SU(Integer(3), CF3, invariant_form=m)
>>> latex(G)
```

(continues on next page)

(continued from previous page)

```
\text{SU}_{3}(\Bold{Q}(\zeta_{3}))\text{ with respect to positive definite␣
↪hermitian form }\left(\begin{array}{rrr}
1 & \zeta_{3} & 0 \\
-\zeta_{3} - 1 & 2 & 0 \\
0 & 0 & 1
\end{array}\right)
```

invariant_form()

Return the hermitian form preserved by the unitary group.

OUTPUT: a square matrix describing the bilinear form

EXAMPLES:

```
sage: SU4 = SU(4, QQ)
sage: SU4.invariant_form()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> SU4 = SU(Integer(4), QQ)
>>> SU4.invariant_form()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

sage.groups.matrix_gps.unitary.**finite_field_sqrt**(*ring*)

Helper function.

OUTPUT: integer q such that `ring` is the finite field with q^2 elements

EXAMPLES:

```
sage: from sage.groups.matrix_gps.unitary import finite_field_sqrt
sage: finite_field_sqrt(GF(4, 'a'))                                          #␣
↪needs sage.rings.finite_rings
2
```

```
>>> from sage.all import *
>>> from sage.groups.matrix_gps.unitary import finite_field_sqrt
>>> finite_field_sqrt(GF(Integer(4), 'a'))                                   ␣
↪    # needs sage.rings.finite_rings
2
```

28.18 Unitary Groups $GU(n, q)$ and $SU(n, q)$ with GAP

class sage.groups.matrix_gps.unitary_gap.**UnitaryMatrixGroup_gap**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *gap_command_string*, *category=None*)

Bases: *UnitaryMatrixGroup_generic*, *NamedMatrixGroup_gap*, *FinitelyGeneratedMatrixGroup_gap*

The general or special unitary group in GAP.

invariant_form()

Return the hermitian form preserved by the unitary group.

OUTPUT: a square matrix describing the bilinear form

EXAMPLES:

```
sage: G32 = GU(3,2)
sage: G32.invariant_form()
[0 0 1]
[0 1 0]
[1 0 0]
```

```
>>> from sage.all import *
>>> G32 = GU(Integer(3),Integer(2))
>>> G32.invariant_form()
[0 0 1]
[0 1 0]
[1 0 0]
```

28.19 Heisenberg Group

AUTHORS:

- Hilder Vitor Lima Pereira (2017-08): initial version

class sage.groups.matrix_gps.heisenberg.**HeisenbergGroup**(*n=1*, *R=0*)

Bases: UniqueRepresentation, *FinitelyGeneratedMatrixGroup_gap*

The Heisenberg group of degree n.

Let R be a ring, and let n be a positive integer. The Heisenberg group of degree n over R is a multiplicative group whose elements are matrices with the following form:

$$\begin{pmatrix} 1 & x^T & z \\ 0 & I_n & y \\ 0 & 0 & 1 \end{pmatrix},$$

where x and y are column vectors in R^n, z is a scalar in R, and I_n is the identity matrix of size n.

INPUT:

- n – the degree of the Heisenberg group
- R – (default: $\mathbf{Z}$) the ring R or a positive integer as a shorthand for the ring $\mathbf{Z}/R\mathbf{Z}$

EXAMPLES:

```
sage: H = groups.matrix.Heisenberg(); H
Heisenberg group of degree 1 over Integer Ring
sage: H.gens()
(
```

(continues on next page)

(continued from previous page)

```
[1 1 0]  [1 0 0]  [1 0 1]
[0 1 0]  [0 1 1]  [0 1 0]
[0 0 1], [0 0 1], [0 0 1]
)
sage: X, Y, Z = H.gens()
sage: Z * X * Y**-1
[ 1  1  0]
[ 0  1 -1]
[ 0  0  1]
sage: X * Y * X**-1 * Y**-1 == Z
True

sage: H = groups.matrix.Heisenberg(R=5); H
Heisenberg group of degree 1 over Ring of integers modulo 5
sage: H = groups.matrix.Heisenberg(n=3, R=13); H
Heisenberg group of degree 3 over Ring of integers modulo 13
```

```
>>> from sage.all import *
>>> H = groups.matrix.Heisenberg(); H
Heisenberg group of degree 1 over Integer Ring
>>> H.gens()
(
[1 1 0]  [1 0 0]  [1 0 1]
[0 1 0]  [0 1 1]  [0 1 0]
[0 0 1], [0 0 1], [0 0 1]
)
>>> X, Y, Z = H.gens()
>>> Z * X * Y**-Integer(1)
[ 1  1  0]
[ 0  1 -1]
[ 0  0  1]
>>> X * Y * X**-Integer(1) * Y**-Integer(1) == Z
True

>>> H = groups.matrix.Heisenberg(R=Integer(5)); H
Heisenberg group of degree 1 over Ring of integers modulo 5
>>> H = groups.matrix.Heisenberg(n=Integer(3), R=Integer(13)); H
Heisenberg group of degree 3 over Ring of integers modulo 13
```

REFERENCES:

- Wikipedia article Heisenberg_group

cardinality()

Return the order of `self`.

EXAMPLES:

```
sage: H = groups.matrix.Heisenberg()
sage: H.order()
+Infinity
sage: H = groups.matrix.Heisenberg(n=4)
sage: H.order()
```

(continues on next page)

(continued from previous page)

```
+Infinity
sage: H = groups.matrix.Heisenberg(R=3)
sage: H.order()
27
sage: H = groups.matrix.Heisenberg(n=2, R=3)
sage: H.order()
243
sage: H = groups.matrix.Heisenberg(n=2, R=GF(4))
sage: H.order()
1024
```

```
>>> from sage.all import *
>>> H = groups.matrix.Heisenberg()
>>> H.order()
+Infinity
>>> H = groups.matrix.Heisenberg(n=Integer(4))
>>> H.order()
+Infinity
>>> H = groups.matrix.Heisenberg(R=Integer(3))
>>> H.order()
27
>>> H = groups.matrix.Heisenberg(n=Integer(2), R=Integer(3))
>>> H.order()
243
>>> H = groups.matrix.Heisenberg(n=Integer(2), R=GF(Integer(4)))
>>> H.order()
1024
```

center()

Return the center of `self`.

This is the subgroup generated by the z, the matrix with a 1 in the upper right corner and along the diagonal.

EXAMPLES:

```
sage: H = groups.matrix.Heisenberg(2)
sage: H.center()
Subgroup with 1 generators (
[1 0 0 1]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
) of Heisenberg group of degree 2 over Integer Ring

sage: H = groups.matrix.Heisenberg(3, 4)
sage: H.center()
Subgroup with 1 generators (
[1 0 0 0 1]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
) of Heisenberg group of degree 3 over Ring of integers modulo 4
```

```
>>> from sage.all import *
>>> H = groups.matrix.Heisenberg(Integer(2))
>>> H.center()
Subgroup with 1 generators (
[1 0 0 1]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
) of Heisenberg group of degree 2 over Integer Ring

>>> H = groups.matrix.Heisenberg(Integer(3), Integer(4))
>>> H.center()
Subgroup with 1 generators (
[1 0 0 0 1]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
) of Heisenberg group of degree 3 over Ring of integers modulo 4
```

order()

Return the order of `self`.

EXAMPLES:

```
sage: H = groups.matrix.Heisenberg()
sage: H.order()
+Infinity
sage: H = groups.matrix.Heisenberg(n=4)
sage: H.order()
+Infinity
sage: H = groups.matrix.Heisenberg(R=3)
sage: H.order()
27
sage: H = groups.matrix.Heisenberg(n=2, R=3)
sage: H.order()
243
sage: H = groups.matrix.Heisenberg(n=2, R=GF(4))
sage: H.order()
1024
```

```
>>> from sage.all import *
>>> H = groups.matrix.Heisenberg()
>>> H.order()
+Infinity
>>> H = groups.matrix.Heisenberg(n=Integer(4))
>>> H.order()
+Infinity
>>> H = groups.matrix.Heisenberg(R=Integer(3))
>>> H.order()
27
>>> H = groups.matrix.Heisenberg(n=Integer(2), R=Integer(3))
>>> H.order()
```

(continues on next page)

(continued from previous page)

```
243
>>> H = groups.matrix.Heisenberg(n=Integer(2), R=GF(Integer(4)))
>>> H.order()
1024
```

28.20 Affine Groups

AUTHORS:

- Volker Braun: initial version

class sage.groups.affine_gps.affine_group.**AffineGroup**(*degree*, *ring*)

Bases: UniqueRepresentation, *Group*

An affine group.

The affine group $\mathrm{Aff}(A)$ (or general affine group) of an affine space A is the group of all invertible affine transformations from the space into itself.

If we let A_V be the affine space of a vector space V (essentially, forgetting what is the origin) then the affine group $\mathrm{Aff}(A_V)$ is the group generated by the general linear group $GL(V)$ together with the translations. Recall that the group of translations acting on A_V is just V itself. The general linear and translation subgroups do not quite commute, and in fact generate the semidirect product

$$\mathrm{Aff}(A_V) = GL(V) \ltimes V.$$

As such, the group elements can be represented by pairs (A, b) of a matrix and a vector. This pair then represents the transformation

$$x \mapsto Ax + b.$$

We can also represent affine transformations as linear transformations by considering $\dim(V) + 1$ dimensional space. We take the affine transformation (A, b) to

$$\begin{pmatrix} A & b \\ 0 & 1 \end{pmatrix}$$

and lifting $x = (x_1, \ldots, x_n)$ to $(x_1, \ldots, x_n, 1)$. Here the $(n + 1)$-th component is always 1, so the linear representations acts on the affine hyperplane $x_{n+1} = 1$ as affine transformations which can be seen directly from the matrix multiplication.

INPUT:

Something that defines an affine space. For example

- An affine space itself:
  - `A` – affine space
- A vector space:
  - `V` – a vector space
- Degree and base ring:
  - `degree` – integer; the degree of the affine group, that is, the dimension of the affine space the group is acting on
  - `ring` – a ring or an integer; the base ring of the affine space. If an integer is given, it must be a prime power and the corresponding finite field is constructed.

- var – (default: 'a') keyword argument to specify the finite field generator name in the case where ring is a prime power

EXAMPLES:

```
sage: F = AffineGroup(3, QQ); F
Affine Group of degree 3 over Rational Field
sage: F(matrix(QQ,[[1,2,3],[4,5,6],[7,8,0]]), vector(QQ,[10,11,12]))
      [1 2 3]     [10]
x |-> [4 5 6] x + [11]
      [7 8 0]     [12]
sage: F([[1,2,3],[4,5,6],[7,8,0]], [10,11,12])
      [1 2 3]     [10]
x |-> [4 5 6] x + [11]
      [7 8 0]     [12]
sage: F([1,2,3,4,5,6,7,8,0], [10,11,12])
      [1 2 3]     [10]
x |-> [4 5 6] x + [11]
      [7 8 0]     [12]
```

```
>>> from sage.all import *
>>> F = AffineGroup(Integer(3), QQ); F
Affine Group of degree 3 over Rational Field
>>> F(matrix(QQ,[[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5),
↪Integer(6)],[Integer(7),Integer(8),Integer(0)]]), vector(QQ,[Integer(10),
↪Integer(11),Integer(12)]))
      [1 2 3]     [10]
x |-> [4 5 6] x + [11]
      [7 8 0]     [12]
>>> F([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5),Integer(6)],
↪[Integer(7),Integer(8),Integer(0)]], [Integer(10),Integer(11),Integer(12)])
      [1 2 3]     [10]
x |-> [4 5 6] x + [11]
      [7 8 0]     [12]
>>> F([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(6),
↪Integer(7),Integer(8),Integer(0)], [Integer(10),Integer(11),Integer(12)])
      [1 2 3]     [10]
x |-> [4 5 6] x + [11]
      [7 8 0]     [12]
```

Instead of specifying the complete matrix/vector information, you can also create special group elements:

```
sage: F.linear([1,2,3,4,5,6,7,8,0])
      [1 2 3]     [0]
x |-> [4 5 6] x + [0]
      [7 8 0]     [0]
sage: F.translation([1,2,3])
      [1 0 0]     [1]
x |-> [0 1 0] x + [2]
      [0 0 1]     [3]
```

```
>>> from sage.all import *
>>> F.linear([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(6),
↪Integer(7),Integer(8),Integer(0)])
```

(continues on next page)

(continued from previous page)

```
       [1 2 3]      [0]
x |-> [4 5 6] x + [0]
       [7 8 0]      [0]
>>> F.translation([Integer(1),Integer(2),Integer(3)])
       [1 0 0]      [1]
x |-> [0 1 0] x + [2]
       [0 0 1]      [3]
```

Some additional ways to create affine groups:

```
sage: A = AffineSpace(2, GF(4,'a'));  A                                        #␣
↪needs sage.rings.finite_rings
Affine Space of dimension 2 over Finite Field in a of size 2^2
sage: G = AffineGroup(A); G                                                    #␣
↪needs sage.rings.finite_rings
Affine Group of degree 2 over Finite Field in a of size 2^2
sage: G is AffineGroup(2,4)  # shorthand                                       #␣
↪needs sage.rings.finite_rings
True

sage: V = ZZ^3;  V
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: AffineGroup(V)
Affine Group of degree 3 over Integer Ring
```

```
>>> from sage.all import *
>>> A = AffineSpace(Integer(2), GF(Integer(4),'a'));  A                        ␣
↪               # needs sage.rings.finite_rings
Affine Space of dimension 2 over Finite Field in a of size 2^2
>>> G = AffineGroup(A); G                                                    #␣
↪needs sage.rings.finite_rings
Affine Group of degree 2 over Finite Field in a of size 2^2
>>> G is AffineGroup(Integer(2),Integer(4))  # shorthand                       ␣
↪               # needs sage.rings.finite_rings
True

>>> V = ZZ**Integer(3);  V
Ambient free module of rank 3 over the principal ideal domain Integer Ring
>>> AffineGroup(V)
Affine Group of degree 3 over Integer Ring
```

REFERENCES:

- Wikipedia article Affine_group

Element

alias of *AffineGroupElement*

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: AffineGroup(6, GF(5)).cardinality()
172882428468750000000000000000

sage: AffineGroup(6, ZZ).cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> AffineGroup(Integer(6), GF(Integer(5))).cardinality()
172882428468750000000000000000

>>> AffineGroup(Integer(6), ZZ).cardinality()
+Infinity
```

degree()

Return the dimension of the affine space.

OUTPUT: integer

EXAMPLES:

```
sage: G = AffineGroup(6, GF(5))
sage: g = G.an_element()
sage: G.degree()
6
sage: G.degree() == g.A().nrows() == g.A().ncols() == g.b().degree()
True
```

```
>>> from sage.all import *
>>> G = AffineGroup(Integer(6), GF(Integer(5)))
>>> g = G.an_element()
>>> G.degree()
6
>>> G.degree() == g.A().nrows() == g.A().ncols() == g.b().degree()
True
```

linear(*A*)

Construct the general linear transformation by `A`.

INPUT:

- `A` – anything that determines a matrix

OUTPUT: the affine group element $x \mapsto Ax$

EXAMPLES:

```
sage: G = AffineGroup(3, GF(5))
sage: G.linear([1,2,3,4,5,6,7,8,0])
      [1 2 3]     [0]
x |-> [4 0 1] x + [0]
      [2 3 0]     [0]
```

```
>>> from sage.all import *
>>> G = AffineGroup(Integer(3), GF(Integer(5)))
>>> G.linear([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6),Integer(7),Integer(8),Integer(0)])
      [1 2 3]     [0]
x |-> [4 0 1] x + [0]
      [2 3 0]     [0]
```

linear_space()

Return the space of the affine transformations represented as linear transformations.

We can represent affine transformations $Ax + b$ as linear transformations by

$$\begin{pmatrix} A & b \\ 0 & 1 \end{pmatrix}$$

and lifting $x = (x_1, \ldots, x_n)$ to $(x_1, \ldots, x_n, 1)$.

See also

- *sage.groups.affine_gps.group_element.AffineGroupElement.matrix()*

EXAMPLES:

```
sage: G = AffineGroup(3, GF(5))
sage: G.linear_space()
Full MatrixSpace of 4 by 4 dense matrices over Finite Field of size 5
```

```
>>> from sage.all import *
>>> G = AffineGroup(Integer(3), GF(Integer(5)))
>>> G.linear_space()
Full MatrixSpace of 4 by 4 dense matrices over Finite Field of size 5
```

matrix_space()

Return the space of matrices representing the general linear transformations.

OUTPUT: the parent of the matrices A defining the affine group element $Ax + b$

EXAMPLES:

```
sage: G = AffineGroup(3, GF(5))
sage: G.matrix_space()
Full MatrixSpace of 3 by 3 dense matrices over Finite Field of size 5
```

```
>>> from sage.all import *
>>> G = AffineGroup(Integer(3), GF(Integer(5)))
>>> G.matrix_space()
Full MatrixSpace of 3 by 3 dense matrices over Finite Field of size 5
```

random_element()

Return a random element of this group.

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: G = AffineGroup(4, GF(3))
sage: G.random_element()   # random
      [2 0 1 2]     [1]
      [2 1 1 2]     [2]
x |-> [1 0 2 2] x + [2]
      [1 1 1 1]     [2]
sage: G.random_element() in G
True
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> G = AffineGroup(Integer(4), GF(Integer(3)))
>>> G.random_element()   # random
      [2 0 1 2]     [1]
      [2 1 1 2]     [2]
x |-> [1 0 2 2] x + [2]
      [1 1 1 1]     [2]
>>> G.random_element() in G
True
```

reflection(*v*)

Construct the Householder reflection.

A Householder reflection (transformation) is the affine transformation corresponding to an elementary reflection at the hyperplane perpendicular to v.

INPUT:

- `v` – a vector, or something that determines a vector

OUTPUT:

The affine group element that is just the Householder transformation (a.k.a. Householder reflection, elementary reflection) at the hyperplane perpendicular to v.

EXAMPLES:

```
sage: G = AffineGroup(3, QQ)
sage: G.reflection([1,0,0])
      [-1  0  0]     [0]
x |-> [ 0  1  0] x + [0]
      [ 0  0  1]     [0]
sage: G.reflection([3,4,-5])
      [ 16/25 -12/25    3/5]     [0]
x |-> [-12/25   9/25    4/5] x + [0]
      [   3/5    4/5      0]     [0]
```

```
>>> from sage.all import *
>>> G = AffineGroup(Integer(3), QQ)
>>> G.reflection([Integer(1),Integer(0),Integer(0)])
      [-1  0  0]     [0]
x |-> [ 0  1  0] x + [0]
      [ 0  0  1]     [0]
>>> G.reflection([Integer(3),Integer(4),-Integer(5)])
```

(continues on next page)

(continued from previous page)

```
       [ 16/25 -12/25     3/5]      [0]
x |-> [-12/25   9/25     4/5] x + [0]
       [   3/5    4/5       0]      [0]
```

some_elements()

Return some elements.

EXAMPLES:

```
sage: # needs sage.libs.gap
sage: G = AffineGroup(4,5)
sage: G.some_elements()
[       [2 0 0 0]      [1]
        [0 1 0 0]      [0]
 x |-> [0 0 1 0] x + [0]
        [0 0 0 1]      [0],
        [2 0 0 0]      [0]
        [0 1 0 0]      [0]
 x |-> [0 0 1 0] x + [0]
        [0 0 0 1]      [0],
        [2 0 0 0]      [...]
        [0 1 0 0]      [...]
 x |-> [0 0 1 0] x + [...]
        [0 0 0 1]      [...]]
sage: all(v.parent() is G for v in G.some_elements())
True

sage: G = AffineGroup(2,QQ)
sage: G.some_elements()
[       [1 0]      [1]
 x |-> [0 1] x + [0],
 ...]
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> G = AffineGroup(Integer(4),Integer(5))
>>> G.some_elements()
[       [2 0 0 0]      [1]
        [0 1 0 0]      [0]
 x |-> [0 0 1 0] x + [0]
        [0 0 0 1]      [0],
        [2 0 0 0]      [0]
        [0 1 0 0]      [0]
 x |-> [0 0 1 0] x + [0]
        [0 0 0 1]      [0],
        [2 0 0 0]      [...]
        [0 1 0 0]      [...]
 x |-> [0 0 1 0] x + [...]
        [0 0 0 1]      [...]]
>>> all(v.parent() is G for v in G.some_elements())
True
```

(continues on next page)

(continued from previous page)

```
>>> G = AffineGroup(Integer(2),QQ)
>>> G.some_elements()
[       [1 0]      [1]
 x |-> [0 1] x + [0],
 ...]
```

translation(*b*)

Construct the translation by b.

INPUT:

- b – anything that determines a vector

OUTPUT: the affine group element $x \mapsto x + b$

EXAMPLES:

```
sage: G = AffineGroup(3, GF(5))
sage: G.translation([1,4,8])
      [1 0 0]     [1]
x |-> [0 1 0] x + [4]
      [0 0 1]     [3]
```

```
>>> from sage.all import *
>>> G = AffineGroup(Integer(3), GF(Integer(5)))
>>> G.translation([Integer(1),Integer(4),Integer(8)])
      [1 0 0]     [1]
x |-> [0 1 0] x + [4]
      [0 0 1]     [3]
```

vector_space()

Return the vector space of the underlying affine space.

EXAMPLES:

```
sage: G = AffineGroup(3, GF(5))
sage: G.vector_space()
Vector space of dimension 3 over Finite Field of size 5
```

```
>>> from sage.all import *
>>> G = AffineGroup(Integer(3), GF(Integer(5)))
>>> G.vector_space()
Vector space of dimension 3 over Finite Field of size 5
```

28.21 Euclidean Groups

AUTHORS:

- Volker Braun: initial version

class sage.groups.affine_gps.euclidean_group.**EuclideanGroup**(*degree*, *ring*)

Bases: *AffineGroup*

A Euclidean group.

The Euclidean group $E(A)$ (or general affine group) of an affine space A is the group of all invertible affine transformations from the space into itself preserving the Euclidean metric.

If we let A_V be the affine space of a vector space V (essentially, forgetting what is the origin) then the Euclidean group $E(A_V)$ is the group generated by the general linear group $SO(V)$ together with the translations. Recall that the group of translations acting on A_V is just V itself. The general linear and translation subgroups do not quite commute, and in fact generate the semidirect product

$$E(A_V) = SO(V) \ltimes V.$$

As such, the group elements can be represented by pairs (A, b) of a matrix and a vector. This pair then represents the transformation

$$x \mapsto Ax + b.$$

We can also represent this as a linear transformation in $\dim(V) + 1$ dimensional space as

$$\begin{pmatrix} A & b \\ 0 & 1 \end{pmatrix}$$

and lifting $x = (x_1, \dots, x_n)$ to $(x_1, \dots, x_n, 1)$.

See also:

- *AffineGroup*

INPUT:

Something that defines an affine space. For example

- An affine space itself:
  - `A` – affine space
- A vector space:
  - `V` – a vector space
- Degree and base ring:
  - `degree` – integer; the degree of the affine group, that is, the dimension of the affine space the group is acting on
  - `ring` – a ring or an integer; the base ring of the affine space. If an integer is given, it must be a prime power and the corresponding finite field is constructed.
  - `var` – (default: `'a'`) keyword argument to specify the finite field generator name in the case where `ring` is a prime power

EXAMPLES:

```
sage: E3 = EuclideanGroup(3, QQ); E3
Euclidean Group of degree 3 over Rational Field
sage: E3(matrix(QQ,[(6/7, -2/7, 3/7), (-2/7, 3/7, 6/7), (3/7, 6/7, -2/7)]),
↪vector(QQ,[10,11,12]))
      [ 6/7 -2/7  3/7]     [10]
x |-> [-2/7  3/7  6/7] x + [11]
      [ 3/7  6/7 -2/7]     [12]
```

(continues on next page)

(continued from previous page)

```
sage: E3([[6/7, -2/7, 3/7], [-2/7, 3/7, 6/7], [3/7, 6/7, -2/7]], [10,11,12])
      [ 6/7 -2/7  3/7]     [10]
x |-> [-2/7  3/7  6/7] x + [11]
      [ 3/7  6/7 -2/7]     [12]
sage: E3([6/7, -2/7, 3/7, -2/7, 3/7, 6/7, 3/7, 6/7, -2/7], [10,11,12])
      [ 6/7 -2/7  3/7]     [10]
x |-> [-2/7  3/7  6/7] x + [11]
      [ 3/7  6/7 -2/7]     [12]
```

```
>>> from sage.all import *
>>> E3 = EuclideanGroup(Integer(3), QQ); E3
Euclidean Group of degree 3 over Rational Field
>>> E3(matrix(QQ,[(Integer(6)/Integer(7), -Integer(2)/Integer(7), Integer(3)/
↪Integer(7)), (-Integer(2)/Integer(7), Integer(3)/Integer(7), Integer(6)/
↪Integer(7)), (Integer(3)/Integer(7), Integer(6)/Integer(7), -Integer(2)/
↪Integer(7))]), vector(QQ,[Integer(10),Integer(11),Integer(12)]))
      [ 6/7 -2/7  3/7]     [10]
x |-> [-2/7  3/7  6/7] x + [11]
      [ 3/7  6/7 -2/7]     [12]
>>> E3([[Integer(6)/Integer(7), -Integer(2)/Integer(7), Integer(3)/Integer(7)], [-
↪Integer(2)/Integer(7), Integer(3)/Integer(7), Integer(6)/Integer(7)],
↪[Integer(3)/Integer(7), Integer(6)/Integer(7), -Integer(2)/Integer(7)]],
↪[Integer(10),Integer(11),Integer(12)])
      [ 6/7 -2/7  3/7]     [10]
x |-> [-2/7  3/7  6/7] x + [11]
      [ 3/7  6/7 -2/7]     [12]
>>> E3([Integer(6)/Integer(7), -Integer(2)/Integer(7), Integer(3)/Integer(7), -
↪Integer(2)/Integer(7), Integer(3)/Integer(7), Integer(6)/Integer(7), Integer(3)/
↪Integer(7), Integer(6)/Integer(7), -Integer(2)/Integer(7)], [Integer(10),
↪Integer(11),Integer(12)])
      [ 6/7 -2/7  3/7]     [10]
x |-> [-2/7  3/7  6/7] x + [11]
      [ 3/7  6/7 -2/7]     [12]
```

Instead of specifying the complete matrix/vector information, you can also create special group elements:

```
sage: E3.linear([6/7, -2/7, 3/7, -2/7, 3/7, 6/7, 3/7, 6/7, -2/7])
      [ 6/7 -2/7  3/7]     [0]
x |-> [-2/7  3/7  6/7] x + [0]
      [ 3/7  6/7 -2/7]     [0]
sage: E3.reflection([4,5,6])
      [ 45/77 -40/77 -48/77]     [0]
x |-> [-40/77  27/77 -60/77] x + [0]
      [-48/77 -60/77   5/77]     [0]
sage: E3.translation([1,2,3])
      [1 0 0]     [1]
x |-> [0 1 0] x + [2]
      [0 0 1]     [3]
```

```
>>> from sage.all import *
>>> E3.linear([Integer(6)/Integer(7), -Integer(2)/Integer(7), Integer(3)/
↪Integer(7), -Integer(2)/Integer(7), Integer(3)/Integer(7), Integer(6)/
```

(continues on next page)

(continued from previous page)

```
↪Integer(7), Integer(3)/Integer(7), Integer(6)/Integer(7), -Integer(2)/
↪Integer(7)])
       [ 6/7 -2/7  3/7]     [0]
x |-> [-2/7  3/7  6/7] x + [0]
       [ 3/7  6/7 -2/7]     [0]
>>> E3.reflection([Integer(4),Integer(5),Integer(6)])
       [ 45/77 -40/77 -48/77]     [0]
x |-> [-40/77  27/77 -60/77] x + [0]
       [-48/77 -60/77   5/77]     [0]
>>> E3.translation([Integer(1),Integer(2),Integer(3)])
       [1 0 0]     [1]
x |-> [0 1 0] x + [2]
       [0 0 1]     [3]
```

Some additional ways to create Euclidean groups:

```
sage: A = AffineSpace(2, GF(4,'a'));  A                                     #␣
↪needs sage.rings.finite_rings
Affine Space of dimension 2 over Finite Field in a of size 2^2
sage: G = EuclideanGroup(A); G                                              #␣
↪needs sage.rings.finite_rings
Euclidean Group of degree 2 over Finite Field in a of size 2^2
sage: G is EuclideanGroup(2,4)   # shorthand                                #␣
↪needs sage.rings.finite_rings
True

sage: V = ZZ^3;  V
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: EuclideanGroup(V)
Euclidean Group of degree 3 over Integer Ring

sage: EuclideanGroup(2, QQ)
Euclidean Group of degree 2 over Rational Field
```

```
>>> from sage.all import *
>>> A = AffineSpace(Integer(2), GF(Integer(4),'a'));  A                     ␣
↪              # needs sage.rings.finite_rings
Affine Space of dimension 2 over Finite Field in a of size 2^2
>>> G = EuclideanGroup(A); G                                              #␣
↪needs sage.rings.finite_rings
Euclidean Group of degree 2 over Finite Field in a of size 2^2
>>> G is EuclideanGroup(Integer(2),Integer(4))   # shorthand                ␣
↪              # needs sage.rings.finite_rings
True

>>> V = ZZ**Integer(3);  V
Ambient free module of rank 3 over the principal ideal domain Integer Ring
>>> EuclideanGroup(V)
Euclidean Group of degree 3 over Integer Ring

>>> EuclideanGroup(Integer(2), QQ)
Euclidean Group of degree 2 over Rational Field
```

REFERENCES:

- Wikipedia article Euclidean_group

random_element()

Return a random element of this group.

EXAMPLES:

```
sage: G = EuclideanGroup(4, GF(3))
sage: G.random_element()   # random
      [2 1 2 1]      [1]
      [1 2 2 1]      [0]
x |-> [2 2 2 2] x + [1]
      [1 1 2 2]      [2]
sage: G.random_element() in G
True
```

```
>>> from sage.all import *
>>> G = EuclideanGroup(Integer(4), GF(Integer(3)))
>>> G.random_element()   # random
      [2 1 2 1]      [1]
      [1 2 2 1]      [0]
x |-> [2 2 2 2] x + [1]
      [1 1 2 2]      [2]
>>> G.random_element() in G
True
```

28.22 Elements of Affine Groups

The class in this module is used to represent the elements of *AffineGroup()* and its subgroups.

EXAMPLES:

```
sage: F = AffineGroup(3, QQ)
sage: F([1,2,3,4,5,6,7,8,0], [10,11,12])
      [1 2 3]      [10]
x |-> [4 5 6] x + [11]
      [7 8 0]      [12]

sage: G = AffineGroup(2, ZZ)
sage: g = G([[1,1],[0,1]], [1,0])
sage: h = G([[1,2],[0,1]], [0,1])
sage: g*h
      [1 3]      [2]
x |-> [0 1] x + [1]
sage: h*g
      [1 3]      [1]
x |-> [0 1] x + [1]
sage: g*h != h*g
True
```

```
>>> from sage.all import *
>>> F = AffineGroup(Integer(3), QQ)
```

(continues on next page)

(continued from previous page)

```
>>> F([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(6),Integer(7),
↪Integer(8),Integer(0)], [Integer(10),Integer(11),Integer(12)])
      [1 2 3]     [10]
x |-> [4 5 6] x + [11]
      [7 8 0]     [12]

>>> G = AffineGroup(Integer(2), ZZ)
>>> g = G([[Integer(1),Integer(1)],[Integer(0),Integer(1)]], [Integer(1),Integer(0)])
>>> h = G([[Integer(1),Integer(2)],[Integer(0),Integer(1)]], [Integer(0),Integer(1)])
>>> g*h
      [1 3]     [2]
x |-> [0 1] x + [1]
>>> h*g
      [1 3]     [1]
x |-> [0 1] x + [1]
>>> g*h != h*g
True
```

AUTHORS:

- Volker Braun

class sage.groups.affine_gps.group_element.**AffineGroupElement**(*parent*, *A*, *b=0*, *convert=True*, *check=True*)

Bases: MultiplicativeGroupElement

An affine group element.

INPUT:

- A – an invertible matrix, or something defining a matrix if convert==True
- b – a vector, or something defining a vector if convert==True (default: 0, defining the zero vector)
- parent – the parent affine group
- convert – boolean (default: True); whether to convert A into the correct matrix space and b into the correct vector space
- check – boolean (default: True); whether to do some checks or just accept the input as valid

As a special case, A can be a matrix obtained from *matrix()*, that is, one row and one column larger. In that case, the group element defining that matrix is reconstructed.

OUTPUT: the affine group element $x \mapsto Ax + b$

EXAMPLES:

```
sage: G = AffineGroup(2, GF(3))

sage: # needs sage.libs.gap
sage: g = G.random_element()
sage: type(g)
<class 'sage.groups.affine_gps.affine_group.AffineGroup_with_category.element_
↪class'>
sage: G(g.matrix()) == g
True
```

(continues on next page)

(continued from previous page)

```
sage: G(2)
      [2 0]     [0]
x |-> [0 2] x + [0]
```

```
>>> from sage.all import *
>>> G = AffineGroup(Integer(2), GF(Integer(3)))

>>> # needs sage.libs.gap
>>> g = G.random_element()
>>> type(g)
<class 'sage.groups.affine_gps.affine_group.AffineGroup_with_category.element_
↪class'>
>>> G(g.matrix()) == g
True

>>> G(Integer(2))
      [2 0]     [0]
x |-> [0 2] x + [0]
```

Conversion from a matrix and a matrix group element:

```
sage: M = Matrix(4, 4, [0, 0, -1, 1, 0, -1, 0, 1, -1, 0, 0, 1, 0, 0, 0, 1])
sage: A = AffineGroup(3, ZZ)
sage: A(M)
      [ 0  0 -1]     [1]
x |-> [ 0 -1  0] x + [1]
      [-1  0  0]     [1]
sage: G = MatrixGroup([M])
sage: A(G.0)
      [ 0  0 -1]     [1]
x |-> [ 0 -1  0] x + [1]
      [-1  0  0]     [1]
```

```
>>> from sage.all import *
>>> M = Matrix(Integer(4), Integer(4), [Integer(0), Integer(0), -Integer(1),␣
↪Integer(1), Integer(0), -Integer(1), Integer(0), Integer(1), -Integer(1),␣
↪Integer(0), Integer(0), Integer(1), Integer(0), Integer(0), Integer(0),␣
↪Integer(1)])
>>> A = AffineGroup(Integer(3), ZZ)
>>> A(M)
      [ 0  0 -1]     [1]
x |-> [ 0 -1  0] x + [1]
      [-1  0  0]     [1]
>>> G = MatrixGroup([M])
>>> A(G.gen(0))
      [ 0  0 -1]     [1]
x |-> [ 0 -1  0] x + [1]
      [-1  0  0]     [1]
```

A()

Return the general linear part of an affine group element.

OUTPUT: the matrix A of the affine group element $Ax + b$

EXAMPLES:

```
sage: G = AffineGroup(3, QQ)
sage: g = G([1,2,3,4,5,6,7,8,0], [10,11,12])
sage: A = g.A(); A
[1 2 3]
[4 5 6]
[7 8 0]
sage: A.is_immutable()
True
```

```
>>> from sage.all import *
>>> G = AffineGroup(Integer(3), QQ)
>>> g = G([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(6),
↪Integer(7),Integer(8),Integer(0)], [Integer(10),Integer(11),Integer(12)])
>>> A = g.A(); A
[1 2 3]
[4 5 6]
[7 8 0]
>>> A.is_immutable()
True
```

b()

Return the translation part of an affine group element.

OUTPUT: the vector b of the affine group element $Ax + b$

EXAMPLES:

```
sage: G = AffineGroup(3, QQ)
sage: g = G([1,2,3,4,5,6,7,8,0], [10,11,12])
sage: b = g.b(); b
(10, 11, 12)
sage: b.is_immutable()
True
```

```
>>> from sage.all import *
>>> G = AffineGroup(Integer(3), QQ)
>>> g = G([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(6),
↪Integer(7),Integer(8),Integer(0)], [Integer(10),Integer(11),Integer(12)])
>>> b = g.b(); b
(10, 11, 12)
>>> b.is_immutable()
True
```

list()

Return list representation of `self`.

EXAMPLES:

```
sage: F = AffineGroup(3, QQ)
sage: g = F([1,2,3,4,5,6,7,8,0], [10,11,12])
sage: g
      [1 2 3]     [10]
```

(continues on next page)

(continued from previous page)

```
x |-> [4 5 6] x + [11]
      [7 8 0]     [12]
sage: g.matrix()
[ 1  2  3|10]
[ 4  5  6|11]
[ 7  8  0|12]
[--------+--]
[ 0  0  0| 1]
sage: g.list()
[[1, 2, 3, 10], [4, 5, 6, 11], [7, 8, 0, 12], [0, 0, 0, 1]]
```

```
>>> from sage.all import *
>>> F = AffineGroup(Integer(3), QQ)
>>> g = F([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(6),
↪Integer(7),Integer(8),Integer(0)], [Integer(10),Integer(11),Integer(12)])
>>> g
      [1 2 3]     [10]
x |-> [4 5 6] x + [11]
      [7 8 0]     [12]
>>> g.matrix()
[ 1  2  3|10]
[ 4  5  6|11]
[ 7  8  0|12]
[--------+--]
[ 0  0  0| 1]
>>> g.list()
[[1, 2, 3, 10], [4, 5, 6, 11], [7, 8, 0, 12], [0, 0, 0, 1]]
```

matrix()

Return the standard matrix representation of `self`.

See also

- *AffineGroup.linear_space()*

EXAMPLES:

```
sage: G = AffineGroup(3, GF(7))
sage: g = G([1,2,3,4,5,6,7,8,0], [10,11,12])
sage: g
      [1 2 3]     [3]
x |-> [4 5 6] x + [4]
      [0 1 0]     [5]
sage: g.matrix()
[1 2 3|3]
[4 5 6|4]
[0 1 0|5]
[-----+-]
[0 0 0|1]
sage: parent(g.matrix())
```

(continues on next page)

(continued from previous page)

```
Full MatrixSpace of 4 by 4 dense matrices over Finite Field of size 7
sage: g.matrix() == matrix(g)
True
```

```
>>> from sage.all import *
>>> G = AffineGroup(Integer(3), GF(Integer(7)))
>>> g = G([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(6),
↪Integer(7),Integer(8),Integer(0)], [Integer(10),Integer(11),Integer(12)])
>>> g
      [1 2 3]     [3]
x |-> [4 5 6] x + [4]
      [0 1 0]     [5]
>>> g.matrix()
[1 2 3|3]
[4 5 6|4]
[0 1 0|5]
[-----+-]
[0 0 0|1]
>>> parent(g.matrix())
Full MatrixSpace of 4 by 4 dense matrices over Finite Field of size 7
>>> g.matrix() == matrix(g)
True
```

Composition of affine group elements equals multiplication of the matrices:

```
sage: # needs sage.libs.gap
sage: g1 = G.random_element()
sage: g2 = G.random_element()
sage: g1.matrix() * g2.matrix() == (g1*g2).matrix()
True
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
>>> g1 = G.random_element()
>>> g2 = G.random_element()
>>> g1.matrix() * g2.matrix() == (g1*g2).matrix()
True
```

CHAPTER

TWENTYNINE

LIE GROUPS

29.1 Nilpotent Lie groups

AUTHORS:

- Eero Hakavuori (2018-09-25): initial version of nilpotent Lie groups

class sage.groups.lie_gps.nilpotent_lie_group.**NilpotentLieGroup**(*L*, *name*, ***kwds*)

Bases: *Group*, DifferentiableManifold

A nilpotent Lie group.

INPUT:

- L – the Lie algebra of the Lie group; must be a finite dimensional nilpotent Lie algebra with basis over a topological field, e.g. **Q** or **R**
- name – string; name (symbol) given to the Lie group

Two types of exponential coordinates are defined on any nilpotent Lie group using the basis of the Lie algebra, see *chart_exp1()* and *chart_exp2()*.

EXAMPLES:

Creation of a nilpotent Lie group:

```
sage: L = lie_algebras.Heisenberg(QQ, 1)
sage: G = L.lie_group(); G
Lie group G of Heisenberg algebra of rank 1 over Rational Field
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, Integer(1))
>>> G = L.lie_group(); G
Lie group G of Heisenberg algebra of rank 1 over Rational Field
```

Giving a different name to the group:

```
sage: L.lie_group('H')
Lie group H of Heisenberg algebra of rank 1 over Rational Field
```

```
>>> from sage.all import *
>>> L.lie_group('H')
Lie group H of Heisenberg algebra of rank 1 over Rational Field
```

Elements can be created using the exponential map:

```
sage: p,q,z = L.basis()
sage: g = G.exp(p); g
exp(p1)
sage: h = G.exp(q); h
exp(q1)
```

```
>>> from sage.all import *
>>> p,q,z = L.basis()
>>> g = G.exp(p); g
exp(p1)
>>> h = G.exp(q); h
exp(q1)
```

Lie group multiplication has the usual product syntax:

```
sage: k = g*h; k
exp(p1 + q1 + 1/2*z)
```

```
>>> from sage.all import *
>>> k = g*h; k
exp(p1 + q1 + 1/2*z)
```

The identity element is given by `one()`:

```
sage: e = G.one(); e
exp(0)
sage: e*k == k and k*e == k
True
```

```
>>> from sage.all import *
>>> e = G.one(); e
exp(0)
>>> e*k == k and k*e == k
True
```

The default coordinate system is exponential coordinates of the first kind:

```
sage: G.default_chart() == G.chart_exp1()
True
sage: G.chart_exp1()
Chart (G, (x_0, x_1, x_2))
```

```
>>> from sage.all import *
>>> G.default_chart() == G.chart_exp1()
True
>>> G.chart_exp1()
Chart (G, (x_0, x_1, x_2))
```

Changing the default coordinates to exponential coordinates of the second kind will change how elements are printed:

```
sage: G.set_default_chart(G.chart_exp2())
sage: k
exp(z)exp(q1)exp(p1)
sage: G.set_default_chart(G.chart_exp1())
sage: k
exp(p1 + q1 + 1/2*z)
```

```
>>> from sage.all import *
>>> G.set_default_chart(G.chart_exp2())
>>> k
exp(z)exp(q1)exp(p1)
>>> G.set_default_chart(G.chart_exp1())
>>> k
exp(p1 + q1 + 1/2*z)
```

The frames of left- or right-invariant vector fields are created using `left_invariant_frame()` and `right_invariant_frame()`:

```
sage: X = G.left_invariant_frame(); X
Vector frame (G, (X_0,X_1,X_2))
sage: X[0]
Vector field X_0 on the Lie group G of Heisenberg algebra of rank 1 over Rational␣
↪Field
```

```
>>> from sage.all import *
>>> X = G.left_invariant_frame(); X
Vector frame (G, (X_0,X_1,X_2))
>>> X[Integer(0)]
Vector field X_0 on the Lie group G of Heisenberg algebra of rank 1 over Rational␣
↪Field
```

A vector field can be displayed with respect to a coordinate frame:

```
sage: exp1_frame = G.chart_exp1().frame()
sage: exp2_frame = G.chart_exp2().frame()
sage: X[0].display(exp1_frame)
X_0 = ∂/∂x_0 - 1/2*x_1 ∂/∂x_2
sage: X[0].display(exp2_frame)
X_0 = ∂/∂y_0
sage: X[1].display(exp1_frame)
X_1 = ∂/∂x_1 + 1/2*x_0 ∂/∂x_2
sage: X[1].display(exp2_frame)
X_1 = ∂/∂y_1 + x_0 ∂/∂y_2
```

```
>>> from sage.all import *
>>> exp1_frame = G.chart_exp1().frame()
>>> exp2_frame = G.chart_exp2().frame()
>>> X[Integer(0)].display(exp1_frame)
X_0 = ∂/∂x_0 - 1/2*x_1 ∂/∂x_2
>>> X[Integer(0)].display(exp2_frame)
X_0 = ∂/∂y_0
>>> X[Integer(1)].display(exp1_frame)
```

(continues on next page)

(continued from previous page)

```
X_1 = ∂/∂x_1 + 1/2*x_0 ∂/∂x_2
>>> X[Integer(1)].display(exp2_frame)
X_1 = ∂/∂y_1 + x_0 ∂/∂y_2
```

Defining a left translation by a generic point:

```
sage: g = G.point([var('a'), var('b'), var('c')]); g
exp(a*p1 + b*q1 + c*z)
sage: L_g = G.left_translation(g); L_g
Diffeomorphism of the Lie group G of Heisenberg algebra of rank 1 over Rational␣
↪Field
sage: L_g.display()
G → G
   (x_0, x_1, x_2) ↦ (a + x_0, b + x_1, -1/2*b*x_0 + 1/2*a*x_1 + c + x_2)
   (x_0, x_1, x_2) ↦ (y_0, y_1, y_2) = (a + x_0, b + x_1,
                                        1/2*a*b + 1/2*(2*a + x_0)*x_1 + c + x_2)
   (y_0, y_1, y_2) ↦ (x_0, x_1, x_2) = (a + y_0, b + y_1,
                                        -1/2*b*y_0 + 1/2*(a - y_0)*y_1 + c + y_2)
   (y_0, y_1, y_2) ↦ (a + y_0, b + y_1, 1/2*a*b + a*y_1 + c + y_2)
```

```
>>> from sage.all import *
>>> g = G.point([var('a'), var('b'), var('c')]); g
exp(a*p1 + b*q1 + c*z)
>>> L_g = G.left_translation(g); L_g
Diffeomorphism of the Lie group G of Heisenberg algebra of rank 1 over Rational␣
↪Field
>>> L_g.display()
G → G
   (x_0, x_1, x_2) ↦ (a + x_0, b + x_1, -1/2*b*x_0 + 1/2*a*x_1 + c + x_2)
   (x_0, x_1, x_2) ↦ (y_0, y_1, y_2) = (a + x_0, b + x_1,
                                        1/2*a*b + 1/2*(2*a + x_0)*x_1 + c + x_2)
   (y_0, y_1, y_2) ↦ (x_0, x_1, x_2) = (a + y_0, b + y_1,
                                        -1/2*b*y_0 + 1/2*(a - y_0)*y_1 + c + y_2)
   (y_0, y_1, y_2) ↦ (a + y_0, b + y_1, 1/2*a*b + a*y_1 + c + y_2)
```

Verifying the left-invariance of the left-invariant frame:

```
sage: x = G(G.chart_exp1()[:])
sage: L_g.differential(x)(X[0].at(x)) == X[0].at(L_g(x))
True
sage: L_g.differential(x)(X[1].at(x)) == X[1].at(L_g(x))
True
sage: L_g.differential(x)(X[2].at(x)) == X[2].at(L_g(x))
True
```

```
>>> from sage.all import *
>>> x = G(G.chart_exp1()[:])
>>> L_g.differential(x)(X[Integer(0)].at(x)) == X[Integer(0)].at(L_g(x))
True
>>> L_g.differential(x)(X[Integer(1)].at(x)) == X[Integer(1)].at(L_g(x))
True
>>> L_g.differential(x)(X[Integer(2)].at(x)) == X[Integer(2)].at(L_g(x))
```

(continues on next page)

(continued from previous page)

```
True
```

An element of the Lie algebra can be extended to a left or right invariant vector field:

```
sage: X_L = G.left_invariant_extension(p + 3*q); X_L
Vector field p1 + 3*q1 on the Lie group G of Heisenberg algebra of rank 1 over␣
↪Rational Field
sage: X_L.display(exp1_frame)
p1 + 3*q1 = ∂/∂x_0 + 3 ∂/∂x_1 + (3/2*x_0 - 1/2*x_1) ∂/∂x_2
sage: X_R = G.right_invariant_extension(p + 3*q)
sage: X_R.display(exp1_frame)
p1 + 3*q1 = ∂/∂x_0 + 3 ∂/∂x_1 + (-3/2*x_0 + 1/2*x_1) ∂/∂x_2
```

```
>>> from sage.all import *
>>> X_L = G.left_invariant_extension(p + Integer(3)*q); X_L
Vector field p1 + 3*q1 on the Lie group G of Heisenberg algebra of rank 1 over␣
↪Rational Field
>>> X_L.display(exp1_frame)
p1 + 3*q1 = ∂/∂x_0 + 3 ∂/∂x_1 + (3/2*x_0 - 1/2*x_1) ∂/∂x_2
>>> X_R = G.right_invariant_extension(p + Integer(3)*q)
>>> X_R.display(exp1_frame)
p1 + 3*q1 = ∂/∂x_0 + 3 ∂/∂x_1 + (-3/2*x_0 + 1/2*x_1) ∂/∂x_2
```

The nilpotency step of the Lie group is the nilpotency step of its algebra. Nilpotency for Lie groups means that group commutators that are longer than the nilpotency step vanish:

```
sage: G.step()
2
sage: g = G.exp(p); h = G.exp(q)
sage: c = g*h*~g*~h; c
exp(z)
sage: g*c*~g*~c
exp(0)
```

```
>>> from sage.all import *
>>> G.step()
2
>>> g = G.exp(p); h = G.exp(q)
>>> c = g*h*~g*~h; c
exp(z)
>>> g*c*~g*~c
exp(0)
```

class Element(*parent*, ***kwds*)

Bases: `ManifoldPoint`, `MultiplicativeGroupElement`

A base class for an element of a Lie group.

EXAMPLES:

Elements of the group are printed in the default exponential coordinates:

```
sage: L.<X,Y,Z> = LieAlgebra(QQ, 2, step=2)
sage: G = L.lie_group()
sage: g = G.exp(2*X + 3*Z); g
exp(2*X + 3*Z)
sage: h = G.point([ var('a'), var('b'), 0]); h
exp(a*X + b*Y)
sage: G.set_default_chart(G.chart_exp2())
sage: g
exp(3*Z)exp(2*X)
sage: h
exp(1/2*a*b*Z)exp(b*Y)exp(a*X)
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(2), names=('X', 'Y', 'Z',));␣
↪(X, Y, Z,) = L._first_ngens(3)
>>> G = L.lie_group()
>>> g = G.exp(Integer(2)*X + Integer(3)*Z); g
exp(2*X + 3*Z)
>>> h = G.point([ var('a'), var('b'), Integer(0)]); h
exp(a*X + b*Y)
>>> G.set_default_chart(G.chart_exp2())
>>> g
exp(3*Z)exp(2*X)
>>> h
exp(1/2*a*b*Z)exp(b*Y)exp(a*X)
```

Multiplication of two elements uses the usual product syntax:

```
sage: G.exp(Y)*G.exp(X)
exp(Y)exp(X)
sage: G.exp(X)*G.exp(Y)
exp(Z)exp(Y)exp(X)
sage: G.set_default_chart(G.chart_exp1())
sage: G.exp(X)*G.exp(Y)
exp(X + Y + 1/2*Z)
```

```
>>> from sage.all import *
>>> G.exp(Y)*G.exp(X)
exp(Y)exp(X)
>>> G.exp(X)*G.exp(Y)
exp(Z)exp(Y)exp(X)
>>> G.set_default_chart(G.chart_exp1())
>>> G.exp(X)*G.exp(Y)
exp(X + Y + 1/2*Z)
```

adjoint(*g*)

Return the adjoint map as an automorphism of the Lie algebra of `self`.

INPUT:

- `g` – an element of `self`

For a Lie group element g, the adjoint map Ad_g is the map on the Lie algebra $\mathfrak{g}$ given by the differential of the conjugation by g at the identity.

If the Lie algebra of `self` does not admit symbolic coefficients, the adjoint is not in general defined for abstract points.

EXAMPLES:

An example of an adjoint map:

```
sage: L = LieAlgebra(QQ, 2, step=3)
sage: G = L.lie_group()
sage: g = G.exp(L.basis().list()[0]); g
exp(X_1)
sage: Ad_g = G.adjoint(g); Ad_g
Lie algebra endomorphism of Free Nilpotent Lie algebra on 5
generators (X_1, X_2, X_12, X_112, X_122) over Rational Field
  Defn: X_1 |--> X_1
        X_2 |--> X_2 + X_12 + 1/2*X_112
        X_12 |--> X_12 + X_112
        X_112 |--> X_112
        X_122 |--> X_122
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(3))
>>> G = L.lie_group()
>>> g = G.exp(L.basis().list()[Integer(0)]); g
exp(X_1)
>>> Ad_g = G.adjoint(g); Ad_g
Lie algebra endomorphism of Free Nilpotent Lie algebra on 5
generators (X_1, X_2, X_12, X_112, X_122) over Rational Field
  Defn: X_1 |--> X_1
        X_2 |--> X_2 + X_12 + 1/2*X_112
        X_12 |--> X_12 + X_112
        X_112 |--> X_112
        X_122 |--> X_122
```

Usually the adjoint map of a symbolic point is not defined:

```
sage: L = LieAlgebra(QQ, 2, step=2)
sage: G = L.lie_group()
sage: g = G.point([var('a'), var('b'), var('c')]); g
exp(a*X_1 + b*X_2 + c*X_12)
sage: G.adjoint(g)
Traceback (most recent call last):
...
TypeError: unable to convert -b to a rational
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(2))
>>> G = L.lie_group()
>>> g = G.point([var('a'), var('b'), var('c')]); g
exp(a*X_1 + b*X_2 + c*X_12)
>>> G.adjoint(g)
Traceback (most recent call last):
...
TypeError: unable to convert -b to a rational
```

However, if the adjoint map is independent from the symbolic terms, the map is still well defined:

```
sage: g = G.point([0, 0, var('a')]); g
exp(a*X_12)
sage: G.adjoint(g)
Lie algebra endomorphism of Free Nilpotent Lie algebra on 3 generators (X_1,␣
↪X_2, X_12) over Rational Field
  Defn: X_1 |--> X_1
        X_2 |--> X_2
        X_12 |--> X_12
```

```
>>> from sage.all import *
>>> g = G.point([Integer(0), Integer(0), var('a')]); g
exp(a*X_12)
>>> G.adjoint(g)
Lie algebra endomorphism of Free Nilpotent Lie algebra on 3 generators (X_1,␣
↪X_2, X_12) over Rational Field
  Defn: X_1 |--> X_1
        X_2 |--> X_2
        X_12 |--> X_12
```

chart_exp1()

Return the chart of exponential coordinates of the first kind.

Exponential coordinates of the first kind are

$$\exp(x_1 X_1 + \cdots + x_n X_n) \mapsto (x_1, \ldots, x_n).$$

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 2, step=2)
sage: G = L.lie_group()
sage: G.chart_exp1()
Chart (G, (x_1, x_2, x_12))
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(2))
>>> G = L.lie_group()
>>> G.chart_exp1()
Chart (G, (x_1, x_2, x_12))
```

chart_exp2()

Return the chart of exponential coordinates of the second kind.

Exponential coordinates of the second kind are

$$\exp(x_n X_n) \cdots \exp(x_1 X_1) \mapsto (x_1, \ldots, x_n).$$

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 2, step=2)
sage: G = L.lie_group()
sage: G.chart_exp2()
Chart (G, (y_1, y_2, y_12))
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(2))
>>> G = L.lie_group()
>>> G.chart_exp2()
Chart (G, (y_1, y_2, y_12))
```

conjugation(*g*)

Return the conjugation by `g` as an automorphism of `self`.

The conjugation by g on a Lie group G is the map

$$G \to G, \qquad h \mapsto ghg^{-1}.$$

INPUT:

- `g` – an element of `self`

EXAMPLES:

A generic conjugation in the Heisenberg group:

```
sage: H = lie_algebras.Heisenberg(QQ, 1)
sage: p,q,z = H.basis()
sage: G = H.lie_group()
sage: g = G.point([var('a'), var('b'), var('c')])
sage: C_g = G.conjugation(g); C_g
Diffeomorphism of the Lie group G of Heisenberg algebra of rank 1 over␣
↪Rational Field
sage: C_g.display(chart1=G.chart_exp1(), chart2=G.chart_exp1())
G → G
   (x_0, x_1, x_2) ↦ (x_0, x_1, -b*x_0 + a*x_1 + x_2)
```

```
>>> from sage.all import *
>>> H = lie_algebras.Heisenberg(QQ, Integer(1))
>>> p,q,z = H.basis()
>>> G = H.lie_group()
>>> g = G.point([var('a'), var('b'), var('c')])
>>> C_g = G.conjugation(g); C_g
Diffeomorphism of the Lie group G of Heisenberg algebra of rank 1 over␣
↪Rational Field
>>> C_g.display(chart1=G.chart_exp1(), chart2=G.chart_exp1())
G → G
   (x_0, x_1, x_2) ↦ (x_0, x_1, -b*x_0 + a*x_1 + x_2)
```

exp(*X*)

Return the group element $exp(X)$.

INPUT:

- `X` – an element of the Lie algebra of `self`

EXAMPLES:

```
sage: L.<X,Y,Z> = LieAlgebra(QQ, 2, step=2)
sage: G = L.lie_group()
sage: G.exp(X)
```

(continues on next page)

(continued from previous page)

```
exp(X)
sage: G.exp(Y)
exp(Y)
sage: G.exp(X + Y)
exp(X + Y)
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(2), names=('X', 'Y', 'Z',));
↪(X, Y, Z,) = L._first_ngens(3)
>>> G = L.lie_group()
>>> G.exp(X)
exp(X)
>>> G.exp(Y)
exp(Y)
>>> G.exp(X + Y)
exp(X + Y)
```

gens()

Return a tuple of elements whose one-parameter subgroups generate the Lie group.

EXAMPLES:

```
sage: L = lie_algebras.Heisenberg(QQ, 1)
sage: G = L.lie_group()
sage: G.gens()
(exp(p1), exp(q1), exp(z))
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, Integer(1))
>>> G = L.lie_group()
>>> G.gens()
(exp(p1), exp(q1), exp(z))
```

left_invariant_extension(*X*, *name=None*)

Return the left-invariant vector field that has the value `X` at the identity.

INPUT:

- `X` – an element of the Lie algebra of `self`
- `name` – (optional) a string to use as a name for the vector field; if nothing is given, the name of the vector `X` is used

EXAMPLES:

A left-invariant extension in the Heisenberg group:

```
sage: L = lie_algebras.Heisenberg(QQ, 1)
sage: p, q, z = L.basis()
sage: H = L.lie_group('H')
sage: X = H.left_invariant_extension(p); X
Vector field p1 on the Lie group H of Heisenberg algebra of rank 1 over
↪Rational Field
sage: X.display(H.chart_exp1().frame())
p1 = ∂/∂x_0 - 1/2*x_1 ∂/∂x_2
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, Integer(1))
>>> p, q, z = L.basis()
>>> H = L.lie_group('H')
>>> X = H.left_invariant_extension(p); X
Vector field p1 on the Lie group H of Heisenberg algebra of rank 1 over
↪Rational Field
>>> X.display(H.chart_exp1().frame())
p1 = ∂/∂x_0 - 1/2*x_1 ∂/∂x_2
```

Default vs. custom naming for the invariant vector field:

```
sage: Y = H.left_invariant_extension(p + q); Y
Vector field p1 + q1 on the Lie group H of Heisenberg algebra of rank 1 over
↪Rational Field
sage: Z = H.left_invariant_extension(p + q, 'Z'); Z
Vector field Z on the Lie group H of Heisenberg algebra of rank 1 over
↪Rational Field
```

```
>>> from sage.all import *
>>> Y = H.left_invariant_extension(p + q); Y
Vector field p1 + q1 on the Lie group H of Heisenberg algebra of rank 1 over
↪Rational Field
>>> Z = H.left_invariant_extension(p + q, 'Z'); Z
Vector field Z on the Lie group H of Heisenberg algebra of rank 1 over
↪Rational Field
```

left_invariant_frame(***kwds*)

Return the frame of left-invariant vector fields of `self`.

The labeling of the frame and the dual frame can be customized using keyword parameters as described in `sage.manifolds.differentiable.manifold.DifferentiableManifold.vector_frame()`.

EXAMPLES:

The default left-invariant frame:

```
sage: L = LieAlgebra(QQ, 2, step=2)
sage: G = L.lie_group()
sage: livf = G.left_invariant_frame(); livf
Vector frame (G, (X_1,X_2,X_12))
sage: coord_frame = G.chart_exp1().frame()
sage: livf[0].display(coord_frame)
X_1 = ∂/∂x_1 - 1/2*x_2 ∂/∂x_12
sage: livf[1].display(coord_frame)
X_2 = ∂/∂x_2 + 1/2*x_1 ∂/∂x_12
sage: livf[2].display(coord_frame)
X_12 = ∂/∂x_12
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(2))
>>> G = L.lie_group()
>>> livf = G.left_invariant_frame(); livf
Vector frame (G, (X_1,X_2,X_12))
```

(continues on next page)

(continued from previous page)

```
>>> coord_frame = G.chart_exp1().frame()
>>> livf[Integer(0)].display(coord_frame)
X_1 = ∂/∂x_1 - 1/2*x_2 ∂/∂x_12
>>> livf[Integer(1)].display(coord_frame)
X_2 = ∂/∂x_2 + 1/2*x_1 ∂/∂x_12
>>> livf[Integer(2)].display(coord_frame)
X_12 = ∂/∂x_12
```

Examples of custom labeling for the frame:

```
sage: G.left_invariant_frame(symbol='Y')
Vector frame (G, (Y_1,Y_2,Y_12))
sage: G.left_invariant_frame(symbol='Z', indices=None)
Vector frame (G, (Z_0,Z_1,Z_2))
sage: G.left_invariant_frame(symbol='W', indices=('a','b','c'))
Vector frame (G, (W_a,W_b,W_c))
```

```
>>> from sage.all import *
>>> G.left_invariant_frame(symbol='Y')
Vector frame (G, (Y_1,Y_2,Y_12))
>>> G.left_invariant_frame(symbol='Z', indices=None)
Vector frame (G, (Z_0,Z_1,Z_2))
>>> G.left_invariant_frame(symbol='W', indices=('a','b','c'))
Vector frame (G, (W_a,W_b,W_c))
```

left_translation(*g*)

Return the left translation by `g` as an automorphism of `self`.

The left translation by g on a Lie group G is the map

$$G \to G, \qquad h \mapsto gh.$$

INPUT:

- `g` – an element of `self`

EXAMPLES:

A left translation in the Heisenberg group:

```
sage: H = lie_algebras.Heisenberg(QQ, 1)
sage: p,q,z = H.basis()
sage: G = H.lie_group()
sage: g = G.exp(p)
sage: L_g = G.left_translation(g); L_g
Diffeomorphism of the Lie group G of Heisenberg algebra of rank 1 over␣
↪Rational Field
sage: L_g.display(chart1=G.chart_exp1(), chart2=G.chart_exp1())
G → G
   (x_0, x_1, x_2) ↦ (x_0 + 1, x_1, 1/2*x_1 + x_2)
```

```
>>> from sage.all import *
>>> H = lie_algebras.Heisenberg(QQ, Integer(1))
>>> p,q,z = H.basis()
```

(continues on next page)

(continued from previous page)

```
>>> G = H.lie_group()
>>> g = G.exp(p)
>>> L_g = G.left_translation(g); L_g
Diffeomorphism of the Lie group G of Heisenberg algebra of rank 1 over␣
↪Rational Field
>>> L_g.display(chart1=G.chart_exp1(), chart2=G.chart_exp1())
G → G
    (x_0, x_1, x_2) ↦ (x_0 + 1, x_1, 1/2*x_1 + x_2)
```

Left translation by a generic element:

```
sage: h = G.point([var('a'), var('b'), var('c')])
sage: L_h = G.left_translation(h)
sage: L_h.display(chart1=G.chart_exp1(), chart2=G.chart_exp1())
G → G
   (x_0, x_1, x_2) ↦ (a + x_0, b + x_1, -1/2*b*x_0 + 1/2*a*x_1 + c + x_2)
```

```
>>> from sage.all import *
>>> h = G.point([var('a'), var('b'), var('c')])
>>> L_h = G.left_translation(h)
>>> L_h.display(chart1=G.chart_exp1(), chart2=G.chart_exp1())
G → G
   (x_0, x_1, x_2) ↦ (a + x_0, b + x_1, -1/2*b*x_0 + 1/2*a*x_1 + c + x_2)
```

lie_algebra()

Return the Lie algebra of `self`.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 2, step=2)
sage: G = L.lie_group()
sage: G.lie_algebra() == L
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(2))
>>> G = L.lie_group()
>>> G.lie_algebra() == L
True
```

livf(***kwds*)

Return the frame of left-invariant vector fields of `self`.

The labeling of the frame and the dual frame can be customized using keyword parameters as described in `sage.manifolds.differentiable.manifold.DifferentiableManifold.vector_frame()`.

EXAMPLES:

The default left-invariant frame:

```
sage: L = LieAlgebra(QQ, 2, step=2)
sage: G = L.lie_group()
sage: livf = G.left_invariant_frame(); livf
Vector frame (G, (X_1,X_2,X_12))
```

(continues on next page)

(continued from previous page)

```
sage: coord_frame = G.chart_exp1().frame()
sage: livf[0].display(coord_frame)
X_1 = ∂/∂x_1 - 1/2*x_2 ∂/∂x_12
sage: livf[1].display(coord_frame)
X_2 = ∂/∂x_2 + 1/2*x_1 ∂/∂x_12
sage: livf[2].display(coord_frame)
X_12 = ∂/∂x_12
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(2))
>>> G = L.lie_group()
>>> livf = G.left_invariant_frame(); livf
Vector frame (G, (X_1,X_2,X_12))
>>> coord_frame = G.chart_exp1().frame()
>>> livf[Integer(0)].display(coord_frame)
X_1 = ∂/∂x_1 - 1/2*x_2 ∂/∂x_12
>>> livf[Integer(1)].display(coord_frame)
X_2 = ∂/∂x_2 + 1/2*x_1 ∂/∂x_12
>>> livf[Integer(2)].display(coord_frame)
X_12 = ∂/∂x_12
```

Examples of custom labeling for the frame:

```
sage: G.left_invariant_frame(symbol='Y')
Vector frame (G, (Y_1,Y_2,Y_12))
sage: G.left_invariant_frame(symbol='Z', indices=None)
Vector frame (G, (Z_0,Z_1,Z_2))
sage: G.left_invariant_frame(symbol='W', indices=('a','b','c'))
Vector frame (G, (W_a,W_b,W_c))
```

```
>>> from sage.all import *
>>> G.left_invariant_frame(symbol='Y')
Vector frame (G, (Y_1,Y_2,Y_12))
>>> G.left_invariant_frame(symbol='Z', indices=None)
Vector frame (G, (Z_0,Z_1,Z_2))
>>> G.left_invariant_frame(symbol='W', indices=('a','b','c'))
Vector frame (G, (W_a,W_b,W_c))
```

log(*x*)

Return the logarithm of the element `x` of `self`.

INPUT:

- `x` – an element of `self`

The logarithm is by definition the inverse of *exp()*.

If the Lie algebra of `self` does not admit symbolic coefficients, the logarithm is not defined for abstract, i.e. symbolic, points.

EXAMPLES:

The logarithm is the inverse of the exponential:

```
sage: L.<X,Y,Z> = LieAlgebra(QQ, 2, step=2)
sage: G = L.lie_group()
sage: G.log(G.exp(X)) == X
True
sage: G.log(G.exp(X)*G.exp(Y))
X + Y + 1/2*Z
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(2), names=('X', 'Y', 'Z',));␣
↪(X, Y, Z,) = L._first_ngens(3)
>>> G = L.lie_group()
>>> G.log(G.exp(X)) == X
True
>>> G.log(G.exp(X)*G.exp(Y))
X + Y + 1/2*Z
```

The logarithm is not defined for abstract (symbolic) points:

```
sage: g = G.point([var('a'), 1, 2]); g
exp(a*X + Y + 2*Z)
sage: G.log(g)
Traceback (most recent call last):
...
TypeError: unable to convert a to a rational
```

```
>>> from sage.all import *
>>> g = G.point([var('a'), Integer(1), Integer(2)]); g
exp(a*X + Y + 2*Z)
>>> G.log(g)
Traceback (most recent call last):
...
TypeError: unable to convert a to a rational
```

one()

Return the identity element of `self`.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 2, step=4)
sage: G = L.lie_group()
sage: G.one()
exp(0)
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(4))
>>> G = L.lie_group()
>>> G.one()
exp(0)
```

right_invariant_extension(*X*, *name=None*)

Return the right-invariant vector field that has the value `X` at the identity.

INPUT:

- `X` – an element of the Lie algebra of `self`
- `name` – (optional) a string to use as a name for the vector field; if nothing is given, the name of the vector `X` is used

EXAMPLES:

A right-invariant extension in the Heisenberg group:

```
sage: L = lie_algebras.Heisenberg(QQ, 1)
sage: p, q, z = L.basis()
sage: H = L.lie_group('H')
sage: X = H.right_invariant_extension(p); X
Vector field p1 on the Lie group H of Heisenberg algebra of rank 1 over␣
↪Rational Field
sage: X.display(H.chart_exp1().frame())
p1 = ∂/∂x_0 + 1/2*x_1 ∂/∂x_2
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, Integer(1))
>>> p, q, z = L.basis()
>>> H = L.lie_group('H')
>>> X = H.right_invariant_extension(p); X
Vector field p1 on the Lie group H of Heisenberg algebra of rank 1 over␣
↪Rational Field
>>> X.display(H.chart_exp1().frame())
p1 = ∂/∂x_0 + 1/2*x_1 ∂/∂x_2
```

Default vs. custom naming for the invariant vector field:

```
sage: Y = H.right_invariant_extension(p + q); Y
Vector field p1 + q1 on the Lie group H of Heisenberg algebra of rank 1 over␣
↪Rational Field
sage: Z = H.right_invariant_extension(p + q, 'Z'); Z
Vector field Z on the Lie group H of Heisenberg algebra of rank 1 over␣
↪Rational Field
```

```
>>> from sage.all import *
>>> Y = H.right_invariant_extension(p + q); Y
Vector field p1 + q1 on the Lie group H of Heisenberg algebra of rank 1 over␣
↪Rational Field
>>> Z = H.right_invariant_extension(p + q, 'Z'); Z
Vector field Z on the Lie group H of Heisenberg algebra of rank 1 over␣
↪Rational Field
```

right_invariant_frame(***kwds*)

Return the frame of right-invariant vector fields of `self`.

The labeling of the frame and the dual frame can be customized using keyword parameters as described in `sage.manifolds.differentiable.manifold.DifferentiableManifold.vector_frame()`.

EXAMPLES:

The default right-invariant frame:

```
sage: L = LieAlgebra(QQ, 2, step=2)
sage: G = L.lie_group()
sage: rivf = G.right_invariant_frame(); rivf
Vector frame (G, (XR_1,XR_2,XR_12))
sage: coord_frame = G.chart_exp1().frame()
sage: rivf[0].display(coord_frame)
XR_1 = ∂/∂x_1 + 1/2*x_2 ∂/∂x_12
sage: rivf[1].display(coord_frame)
XR_2 = ∂/∂x_2 - 1/2*x_1 ∂/∂x_12
sage: rivf[2].display(coord_frame)
XR_12 = ∂/∂x_12
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(2))
>>> G = L.lie_group()
>>> rivf = G.right_invariant_frame(); rivf
Vector frame (G, (XR_1,XR_2,XR_12))
>>> coord_frame = G.chart_exp1().frame()
>>> rivf[Integer(0)].display(coord_frame)
XR_1 = ∂/∂x_1 + 1/2*x_2 ∂/∂x_12
>>> rivf[Integer(1)].display(coord_frame)
XR_2 = ∂/∂x_2 - 1/2*x_1 ∂/∂x_12
>>> rivf[Integer(2)].display(coord_frame)
XR_12 = ∂/∂x_12
```

Examples of custom labeling for the frame:

```
sage: G.right_invariant_frame(symbol='Y')
Vector frame (G, (Y_1,Y_2,Y_12))
sage: G.right_invariant_frame(symbol='Z', indices=None)
Vector frame (G, (Z_0,Z_1,Z_2))
sage: G.right_invariant_frame(symbol='W', indices=('a','b','c'))
Vector frame (G, (W_a,W_b,W_c))
```

```
>>> from sage.all import *
>>> G.right_invariant_frame(symbol='Y')
Vector frame (G, (Y_1,Y_2,Y_12))
>>> G.right_invariant_frame(symbol='Z', indices=None)
Vector frame (G, (Z_0,Z_1,Z_2))
>>> G.right_invariant_frame(symbol='W', indices=('a','b','c'))
Vector frame (G, (W_a,W_b,W_c))
```

right_translation(*g*)

Return the right translation by `g` as an automorphism of `self`.

The right translation by g on a Lie group G is the map

$$G \to G, \qquad h \mapsto hg.$$

INPUT:

- `g` – an element of `self`

EXAMPLES:

A right translation in the Heisenberg group:

```
sage: H = lie_algebras.Heisenberg(QQ, 1)
sage: p,q,z = H.basis()
sage: G = H.lie_group()
sage: g = G.exp(p)
sage: R_g = G.right_translation(g); R_g
Diffeomorphism of the Lie group G of Heisenberg algebra of rank 1 over␣
↪Rational Field
sage: R_g.display(chart1=G.chart_exp1(), chart2=G.chart_exp1())
G → G
   (x_0, x_1, x_2) ↦ (x_0 + 1, x_1, -1/2*x_1 + x_2)
```

```
>>> from sage.all import *
>>> H = lie_algebras.Heisenberg(QQ, Integer(1))
>>> p,q,z = H.basis()
>>> G = H.lie_group()
>>> g = G.exp(p)
>>> R_g = G.right_translation(g); R_g
Diffeomorphism of the Lie group G of Heisenberg algebra of rank 1 over␣
↪Rational Field
>>> R_g.display(chart1=G.chart_exp1(), chart2=G.chart_exp1())
G → G
   (x_0, x_1, x_2) ↦ (x_0 + 1, x_1, -1/2*x_1 + x_2)
```

Right translation by a generic element:

```
sage: h = G.point([var('a'), var('b'), var('c')])
sage: R_h = G.right_translation(h)
sage: R_h.display(chart1=G.chart_exp1(), chart2=G.chart_exp1())
G → G
   (x_0, x_1, x_2) ↦ (a + x_0, b + x_1, 1/2*b*x_0 - 1/2*a*x_1 + c + x_2)
```

```
>>> from sage.all import *
>>> h = G.point([var('a'), var('b'), var('c')])
>>> R_h = G.right_translation(h)
>>> R_h.display(chart1=G.chart_exp1(), chart2=G.chart_exp1())
G → G
   (x_0, x_1, x_2) ↦ (a + x_0, b + x_1, 1/2*b*x_0 - 1/2*a*x_1 + c + x_2)
```

rivf(***kwds*)

Return the frame of right-invariant vector fields of `self`.

The labeling of the frame and the dual frame can be customized using keyword parameters as described in `sage.manifolds.differentiable.manifold.DifferentiableManifold.vector_frame()`.

EXAMPLES:

The default right-invariant frame:

```
sage: L = LieAlgebra(QQ, 2, step=2)
sage: G = L.lie_group()
sage: rivf = G.right_invariant_frame(); rivf
Vector frame (G, (XR_1,XR_2,XR_12))
sage: coord_frame = G.chart_exp1().frame()
sage: rivf[0].display(coord_frame)
```

(continues on next page)

(continued from previous page)

```
XR_1 = ∂/∂x_1 + 1/2*x_2 ∂/∂x_12
sage: rivf[1].display(coord_frame)
XR_2 = ∂/∂x_2 - 1/2*x_1 ∂/∂x_12
sage: rivf[2].display(coord_frame)
XR_12 = ∂/∂x_12
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(2))
>>> G = L.lie_group()
>>> rivf = G.right_invariant_frame(); rivf
Vector frame (G, (XR_1,XR_2,XR_12))
>>> coord_frame = G.chart_exp1().frame()
>>> rivf[Integer(0)].display(coord_frame)
XR_1 = ∂/∂x_1 + 1/2*x_2 ∂/∂x_12
>>> rivf[Integer(1)].display(coord_frame)
XR_2 = ∂/∂x_2 - 1/2*x_1 ∂/∂x_12
>>> rivf[Integer(2)].display(coord_frame)
XR_12 = ∂/∂x_12
```

Examples of custom labeling for the frame:

```
sage: G.right_invariant_frame(symbol='Y')
Vector frame (G, (Y_1,Y_2,Y_12))
sage: G.right_invariant_frame(symbol='Z', indices=None)
Vector frame (G, (Z_0,Z_1,Z_2))
sage: G.right_invariant_frame(symbol='W', indices=('a','b','c'))
Vector frame (G, (W_a,W_b,W_c))
```

```
>>> from sage.all import *
>>> G.right_invariant_frame(symbol='Y')
Vector frame (G, (Y_1,Y_2,Y_12))
>>> G.right_invariant_frame(symbol='Z', indices=None)
Vector frame (G, (Z_0,Z_1,Z_2))
>>> G.right_invariant_frame(symbol='W', indices=('a','b','c'))
Vector frame (G, (W_a,W_b,W_c))
```

step()

Return the nilpotency step of `self`.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 2, step=4)
sage: G = L.lie_group()
sage: G.step()
4
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(4))
>>> G = L.lie_group()
>>> G.step()
4
```

CHAPTER

THIRTY

PARTITION REFINEMENT

30.1 Canonical augmentation

This module implements a general algorithm for generating isomorphism classes of objects. The class of objects in question must be some kind of structure which can be built up out of smaller objects by a process of augmentation, and for which an automorphism is a permutation in S_n for some n. This process consists of starting with a finite number of "seed objects" and building up to more complicated objects by a sequence of "augmentations." It should be noted that the word "canonical" in the term canonical augmentation is used loosely. Given an object X, one must define a canonical parent $M(X)$, which is essentially an arbitrary choice.

The class of objects in question must satisfy the assumptions made in the module `automorphism_group_canonical_label`, in particular the three custom functions mentioned there must be implemented:

A. `refine_and_return_invariant`:

   Signature:

   `int refine_and_return_invariant(PartitionStack *PS, void *S, int *cells_to_refine_by, int ctrb_len)`

B. `compare_structures`:

   Signature:

   `int compare_structures(int *gamma_1, int *gamma_2, void *S1, void *S2, int degree)`

C. `all_children_are_equivalent`:

   Signature:

   `bint all_children_are_equivalent(PartitionStack *PS, void *S)`

In the following functions there is frequently a mem_err input. This is a pointer to an integer which must be set to a nonzero value in case of an allocation failure. Other functions have an int return value which serves the same purpose. The idea is that if a memory error occurs, the canonical generator should still be able to iterate over the objects already generated before it terminates.

More details about these functions can be found in that module. In addition, several other functions must be implemented, which will make use of the following:

```
ctypedef struct iterator:
    void *data
    void *(*next)(void *data, int *degree, int *mem_err)
```

The following functions must be implemented for each specific type of object to be generated. Each function following which takes a `mem_err` variable as input should make use of this variable.

D. `generate_children`:

Signature:

`int generate_children(void *S, aut_gp_and_can_lab *group, iterator *it)`

This function receives a pointer to an iterator `it`. The iterator has two fields: `data` and `next`. The function `generate_children` should set these two fields, returning 1 to indicate a memory error, or 0 for no error.

The function that `next` points to takes `data` as an argument, and should return a (`void *`) pointer to the next object to be iterated. It also takes a pointer to an int, and must update that int to reflect the degree of each generated object. The objects to be iterated over should satisfy the property that if γ is an automorphism of the parent object S, then for any two child objects C_1, C_2 given by the iterator, it is not the case that $\gamma(C_1) = C_2$, where in the latter γ is appropriately extended if necessary to operate on C_1 and C_2. It is essential for this iterator to handle its own `data`. If the `next` function is called and no suitable object is yielded, a NULL pointer indicates a termination of the iteration. At this point, the data pointed to by the `data` variable should be cleared by the `next` function, because the iterator struct itself will be deallocated.

The `next` function must check `mem_err[0]` before proceeding. If it is nonzero then the function should deallocate the iterator right away and return NULL to end the iteration. This ensures that the canonical augmentation software will finish iterating over the objects found before finishing, and the `mem_err` attribute of the `canonical_generator_data` will reflect this.

The objects which the iterator generates can be thought of as augmentations, which the following function must turn into objects.

E. `apply_augmentation`:

Signature:

`void *apply_augmentation(void *parent, void *aug, void *child, int *degree, bint *mem_err)`

This function takes the `parent`, applies the augmentation `aug` and returns a pointer to the corresponding child object (freeing aug if necessary). Should also update degree[0] to be the degree of the new child.

F. `free_object`:

Signature:

`void free_object(void *child)`

This function is a simple deallocation function for children which are not canonically generated, and therefore rejected in the canonical augmentation process. They should deallocate the contents of `child`.

G. `free_iter_data`:

Signature:

`void free_iter_data(void *data)`

This function deallocates the data part of the iterator which is set up by `generate_children`.

H. `free_aug`:

Signature:

`void free_aug(void *aug)`

This function frees an augmentation as generated by the iterator returned by `generate_children`.

I. `canonical_parent`:

Signature:

`void *canonical_parent(void *child, void *parent, int *permutation, int *degree, bint *mem_err)`

Apply the `permutation` to the `child`, determine an arbitrary but fixed parent, apply the inverse of `permutation` to that parent, and return the resulting object. Must also set the integer `degree` points to the degree of the returned object.

Note

It is a good idea to try to implement an augmentation scheme where the degree of objects on each level of the augmentation tree is constant. The iteration will be more efficient in this case, as the relevant work spaces will never need to be reallocated. Otherwise, one should at least strive to iterate over augmentations in such a way that all children of the same degree are given in the same segment of iteration.

EXAMPLES:

```
sage: import sage.groups.perm_gps.partn_ref.canonical_augmentation
```

```
>>> from sage.all import *
>>> import sage.groups.perm_gps.partn_ref.canonical_augmentation
```

REFERENCE:

- [1] McKay, Brendan D. Isomorph-free exhaustive generation. J Algorithms, Vol. 26 (1998), pp. 306-324.

30.2 Data structures

This module implements basic data structures essential to the rest of the partn_ref module.

REFERENCES:

[1] McKay, Brendan D. Practical Graph Isomorphism. Congressus Numerantium,
Vol. 30 (1981), pp. 45-87.

[2] Fredman, M. and Saks, M. The cell probe complexity of dynamic data
structures. Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, pp. 345–354. May 1989.

[3] Seress, Akos. Permutation Group Algorithms. Cambridge University Press,

2003.

`sage.groups.perm_gps.partn_ref.data_structures.`**`OP_represent`**`(`*n*, *merges*, *perm*`)`

Demonstration and testing.

`sage.groups.perm_gps.partn_ref.data_structures.`**`PS_represent`**`(`*partition*, *splits*`)`

Demonstration and testing.

`sage.groups.perm_gps.partn_ref.data_structures.`**`SC_test_list_perms`**`(`*L*, *n*, *limit*, *gap*, *limit_complain*, *test_contains*`)`

Test that the permutation group generated by list perms in L of degree n is of the correct order, by comparing with GAP. Don't test if the group is of size greater than limit.

30.3 Graph-theoretic partition backtrack functions

EXAMPLES:

```
sage: import sage.groups.perm_gps.partn_ref.refinement_graphs
```

```
>>> from sage.all import *
>>> import sage.groups.perm_gps.partn_ref.refinement_graphs
```

REFERENCE:

- [1] McKay, Brendan D. *Practical Graph Isomorphism*. Congressus Numerantium, Vol. 30 (1981), pp. 45-87.

class sage.groups.perm_gps.partn_ref.refinement_graphs.**GraphStruct**

Bases: object

sage.groups.perm_gps.partn_ref.refinement_graphs.**all_labeled_graphs**(*n*)

Return all labeled graphs on n vertices {0,1,…,n-1}.

Used in classifying isomorphism types (naive approach), and more importantly in benchmarking the search algorithm.

EXAMPLES:

```
sage: from sage.groups.perm_gps.partn_ref.refinement_graphs import all_labeled_
↪graphs
sage: st = sage.groups.perm_gps.partn_ref.refinement_graphs.search_tree
sage: Glist = {}
sage: Giso  = {}
sage: for n in [1..5]:  # long time (4s on sage.math, 2011)
....:  Glist[n] = all_labeled_graphs(n)
....:  Giso[n] = []
....:  for g in Glist[n]:
....:      a, b = st(g, [range(n)])
....:      inn = False
....:      for gi in Giso[n]:
....:          if b == gi:
....:              inn = True
....:      if not inn:
....:          Giso[n].append(b)
sage: for n in Giso:  # long time
....:    print("{} {}".format(n, len(Giso[n])))
1 1
2 2
3 4
4 11
5 34
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.partn_ref.refinement_graphs import all_labeled_
↪graphs
>>> st = sage.groups.perm_gps.partn_ref.refinement_graphs.search_tree
>>> Glist = {}
>>> Giso  = {}
>>> for n in (ellipsis_range(Integer(1),Ellipsis,Integer(5))):  # long time (4s␣
```

(continues on next page)

(continued from previous page)

```
↪on sage.math, 2011)
...   Glist[n] = all_labeled_graphs(n)
...   Giso[n] = []
...   for g in Glist[n]:
...       a, b = st(g, [range(n)])
...       inn = False
...       for gi in Giso[n]:
...           if b == gi:
...               inn = True
...       if not inn:
...           Giso[n].append(b)
>>> for n in Giso:  # long time
...     print("{} {}".format(n, len(Giso[n])))
1 1
2 2
3 4
4 11
5 34
```

sage.groups.perm_gps.partn_ref.refinement_graphs.**coarsest_equitable_refinement**(*G*, *partition*, *directed*)

Return the coarsest equitable refinement of `partition` for `G`.

This is a helper function for the graph function of the same name.

DOCTEST (More thorough testing in `sage/graphs/graph.py`):

```
sage: from sage.groups.perm_gps.partn_ref.refinement_graphs import coarsest_
↪equitable_refinement
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: coarsest_equitable_refinement(SparseGraph(7), [[0], [1,2,3,4], [5,6]], 0)
[[0], [1, 2, 3, 4], [5, 6]]
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.partn_ref.refinement_graphs import coarsest_
↪equitable_refinement
>>> from sage.graphs.base.sparse_graph import SparseGraph
>>> coarsest_equitable_refinement(SparseGraph(Integer(7)), [[Integer(0)],␣
↪[Integer(1),Integer(2),Integer(3),Integer(4)], [Integer(5),Integer(6)]],␣
↪Integer(0))
[[0], [1, 2, 3, 4], [5, 6]]
```

sage.groups.perm_gps.partn_ref.refinement_graphs.**generate_dense_graphs_edge_addition**(*n*, *loops*, *G=None*, *depth=None*, *construct=False*, *indicate_mem_err=True*)

EXAMPLES:

```
sage: from sage.groups.perm_gps.partn_ref.refinement_graphs import generate_dense_
↪graphs_edge_addition
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.partn_ref.refinement_graphs import generate_dense_
↪graphs_edge_addition
```

```
sage: for n in [0..6]:
....:     print(generate_dense_graphs_edge_addition(n,1))
1
2
6
20
90
544
5096
```

```
>>> from sage.all import *
>>> for n in (ellipsis_range(Integer(0),Ellipsis,Integer(6))):
...     print(generate_dense_graphs_edge_addition(n,Integer(1)))
1
2
6
20
90
544
5096
```

```
sage: for n in [0..7]:
....:     print(generate_dense_graphs_edge_addition(n,0))
1
1
2
4
11
34
156
1044
sage: generate_dense_graphs_edge_addition(8,0)  # long time (about 14 seconds at␣
↪2.4 GHz)
12346
```

```
>>> from sage.all import *
>>> for n in (ellipsis_range(Integer(0),Ellipsis,Integer(7))):
...     print(generate_dense_graphs_edge_addition(n,Integer(0)))
1
1
2
4
11
34
```

(continues on next page)

(continued from previous page)

```
156
1044
>>> generate_dense_graphs_edge_addition(Integer(8),Integer(0))  # long time␣
↪(about 14 seconds at 2.4 GHz)
12346
```

sage.groups.perm_gps.partn_ref.refinement_graphs.**generate_dense_graphs_vert_addition**(*n*, *base_G=None*, *construct=False*, *indicate_mem_err=True*)

EXAMPLES:

```
sage: from sage.groups.perm_gps.partn_ref.refinement_graphs import generate_dense_
↪graphs_vert_addition
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.partn_ref.refinement_graphs import generate_dense_
↪graphs_vert_addition
```

```
sage: for n in [0..7]:
....:   generate_dense_graphs_vert_addition(n)
1
2
4
8
19
53
209
1253
sage: generate_dense_graphs_vert_addition(8)  # long time
13599
```

```
>>> from sage.all import *
>>> for n in (ellipsis_range(Integer(0),Ellipsis,Integer(7))):
...   generate_dense_graphs_vert_addition(n)
1
2
4
8
19
53
209
1253
>>> generate_dense_graphs_vert_addition(Integer(8))  # long time
13599
```

sage.groups.perm_gps.partn_ref.refinement_graphs.**get_orbits**(*gens*, *n*)

Compute orbits given a list of generators of a permutation group, in list format.

This is a helper function for automorphism groups of graphs.

DOCTEST (More thorough testing in `sage/graphs/graph.py`):

```
sage: from sage.groups.perm_gps.partn_ref.refinement_graphs import get_orbits
sage: get_orbits([[1,2,3,0,4,5], [0,1,2,3,5,4]], 6)
[[0, 1, 2, 3], [4, 5]]
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.partn_ref.refinement_graphs import get_orbits
>>> get_orbits([[Integer(1),Integer(2),Integer(3),Integer(0),Integer(4),
↪Integer(5)], [Integer(0),Integer(1),Integer(2),Integer(3),Integer(5),
↪Integer(4)]], Integer(6))
[[0, 1, 2, 3], [4, 5]]
```

`sage.groups.perm_gps.partn_ref.refinement_graphs.`**`isomorphic`**`(`*G1*, *G2*, *partn*, *ordering2*, *dig*, *use_indicator_function*, *sparse=False*`)`

Test whether two graphs are isomorphic.

EXAMPLES:

```
sage: from sage.groups.perm_gps.partn_ref.refinement_graphs import isomorphic

sage: G = Graph(2)
sage: H = Graph(2)
sage: isomorphic(G, H, [[0,1]], [0,1], 0, 1)
{0: 0, 1: 1}
sage: isomorphic(G, H, [[0,1]], [0,1], 0, 1)
{0: 0, 1: 1}
sage: isomorphic(G, H, [[0],[1]], [0,1], 0, 1)
{0: 0, 1: 1}
sage: isomorphic(G, H, [[0],[1]], [1,0], 0, 1)
{0: 1, 1: 0}

sage: G = Graph(3)
sage: H = Graph(3)
sage: isomorphic(G, H, [[0,1,2]], [0,1,2], 0, 1)
{0: 0, 1: 1, 2: 2}
sage: G.add_edge(0,1)
sage: isomorphic(G, H, [[0,1,2]], [0,1,2], 0, 1)
False
sage: H.add_edge(1,2)
sage: isomorphic(G, H, [[0,1,2]], [0,1,2], 0, 1)
{0: 1, 1: 2, 2: 0}
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.partn_ref.refinement_graphs import isomorphic

>>> G = Graph(Integer(2))
>>> H = Graph(Integer(2))
>>> isomorphic(G, H, [[Integer(0),Integer(1)]], [Integer(0),Integer(1)],␣
↪Integer(0), Integer(1))
{0: 0, 1: 1}
```

(continues on next page)

(continued from previous page)

```
>>> isomorphic(G, H, [[Integer(0),Integer(1)]], [Integer(0),Integer(1)],␣
↪Integer(0), Integer(1))
{0: 0, 1: 1}
>>> isomorphic(G, H, [[Integer(0)],[Integer(1)]], [Integer(0),Integer(1)],␣
↪Integer(0), Integer(1))
{0: 0, 1: 1}
>>> isomorphic(G, H, [[Integer(0)],[Integer(1)]], [Integer(1),Integer(0)],␣
↪Integer(0), Integer(1))
{0: 1, 1: 0}

>>> G = Graph(Integer(3))
>>> H = Graph(Integer(3))
>>> isomorphic(G, H, [[Integer(0),Integer(1),Integer(2)]], [Integer(0),Integer(1),
↪Integer(2)], Integer(0), Integer(1))
{0: 0, 1: 1, 2: 2}
>>> G.add_edge(Integer(0),Integer(1))
>>> isomorphic(G, H, [[Integer(0),Integer(1),Integer(2)]], [Integer(0),Integer(1),
↪Integer(2)], Integer(0), Integer(1))
False
>>> H.add_edge(Integer(1),Integer(2))
>>> isomorphic(G, H, [[Integer(0),Integer(1),Integer(2)]], [Integer(0),Integer(1),
↪Integer(2)], Integer(0), Integer(1))
{0: 1, 1: 2, 2: 0}
```

sage.groups.perm_gps.partn_ref.refinement_graphs.**orbit_partition**(*gamma*, *list_perm=False*)

Assuming that G is a graph on vertices 0,1,…,n-1, and gamma is an element of SymmetricGroup(n), returns the partition of the vertex set determined by the orbits of gamma, considered as action on the set 1,2,…,n where we take 0 = n. In other words, returns the partition determined by a cyclic representation of gamma.

INPUT:

- `list_perm` – if `True`, assumes `gamma` is a list representing the map $i \mapsto ``gamma``[i]$

EXAMPLES:

```
sage: # needs sage.groups
sage: from sage.groups.perm_gps.partn_ref.refinement_graphs import orbit_partition
sage: G = graphs.PetersenGraph()
sage: S = SymmetricGroup(10)
sage: gamma = S('(10,1,2,3,4)(5,6,7)(8,9)')
sage: orbit_partition(gamma)
[[1, 2, 3, 4, 0], [5, 6, 7], [8, 9]]
sage: gamma = S('(10,5)(1,6)(2,7)(3,8)(4,9)')
sage: orbit_partition(gamma)
[[1, 6], [2, 7], [3, 8], [4, 9], [5, 0]]
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> from sage.groups.perm_gps.partn_ref.refinement_graphs import orbit_partition
>>> G = graphs.PetersenGraph()
>>> S = SymmetricGroup(Integer(10))
>>> gamma = S('(10,1,2,3,4)(5,6,7)(8,9)')
>>> orbit_partition(gamma)
```

(continues on next page)

(continued from previous page)

```
[[1, 2, 3, 4, 0], [5, 6, 7], [8, 9]]
>>> gamma = S('(10,5)(1,6)(2,7)(3,8)(4,9)')
>>> orbit_partition(gamma)
[[1, 6], [2, 7], [3, 8], [4, 9], [5, 0]]
```

`sage.groups.perm_gps.partn_ref.refinement_graphs.`**`random_tests`**`(`*num=10*, *n_max=60*, *perms_per_graph=5*`)`

Test to make sure that C(gamma(G)) == C(G) for random permutations gamma and random graphs G, and that isomorphic returns an isomorphism.

INPUT:

- `num` – run tests for this many graphs
- `n_max` – test graphs with at most this many vertices
- `perms_per_graph` – test each graph with this many random permutations

DISCUSSION:

This code generates num random graphs G on at most n_max vertices. The density of edges is chosen randomly between 0 and 1.

For each graph G generated, we uniformly generate perms_per_graph random permutations and verify that the canonical labels of G and the image of G under the generated permutation are equal, and that the isomorphic function returns an isomorphism.

`sage.groups.perm_gps.partn_ref.refinement_graphs.`**`search_tree`**`(`*G_in*, *partition*, *lab=True*, *dig=False*, *dict_rep=False*, *certificate=False*, *verbosity=0*, *use_indicator_function=True*, *sparse=True*, *base=False*, *order=False*`)`

Compute canonical labels and automorphism groups of graphs.

INPUT:

- `G_in` – a Sage graph
- `partition` – list of lists representing a partition of the vertices
- `lab` – if `True`, compute and return the canonical label in addition to the automorphism group
- `dig` – set to `True` for digraphs and graphs with loops; if `True`, does not use optimizations based on Lemma 2.25 in [1] that are valid only for simple graphs
- `dict_rep` – if `True`, return a dictionary with keys the vertices of the input graph G_in and values elements of the set the permutation group acts on. (The point is that graphs are arbitrarily labelled, often 0..n-1, and permutation groups always act on 1..n. This dictionary maps vertex labels (such as 0..n-1) to the domain of the permutations.)
- `certificate` – if `True`, return the permutation from G to its canonical label
- `verbosity` – currently ignored
- `use_indicator_function` – option to turn off indicator function (`True` is generally faster)
- `sparse` – whether to use sparse or dense representation of the graph (ignored if G is already a CGraph - see sage.graphs.base)
- `base` – whether to return the first sequence of split vertices (used in computing the order of the group)

- order – whether to return the order of the automorphism group

OUTPUT:

Depends on the options. If more than one thing is returned, they are in a tuple in the following order:

- list of generators in list-permutation format – always
- dict – if dict_rep
- graph – if lab
- dict – if certificate
- list – if base
- integer – if order

EXAMPLES:

```
sage: st = sage.groups.perm_gps.partn_ref.refinement_graphs.search_tree
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: from sage.graphs.base.sparse_graph import SparseGraph
```

```
>>> from sage.all import *
>>> st = sage.groups.perm_gps.partn_ref.refinement_graphs.search_tree
>>> from sage.graphs.base.dense_graph import DenseGraph
>>> from sage.graphs.base.sparse_graph import SparseGraph
```

Graphs on zero vertices:

```
sage: G = Graph()
sage: st(G, [[]], order=True)
([], Graph on 0 vertices, 1)
```

```
>>> from sage.all import *
>>> G = Graph()
>>> st(G, [[]], order=True)
([], Graph on 0 vertices, 1)
```

Graphs on one vertex:

```
sage: G = Graph(1)
sage: st(G, [[0]], order=True)
([], Graph on 1 vertex, 1)
```

```
>>> from sage.all import *
>>> G = Graph(Integer(1))
>>> st(G, [[Integer(0)]], order=True)
([], Graph on 1 vertex, 1)
```

Graphs on two vertices:

```
sage: G = Graph(2)
sage: st(G, [[0,1]], order=True)
([[1, 0]], Graph on 2 vertices, 2)
sage: st(G, [[0],[1]], order=True)
([], Graph on 2 vertices, 1)
```

(continues on next page)

(continued from previous page)

```
sage: G.add_edge(0,1)
sage: st(G, [[0,1]], order=True)
([[1, 0]], Graph on 2 vertices, 2)
sage: st(G, [[0],[1]], order=True)
([], Graph on 2 vertices, 1)
```

```
>>> from sage.all import *
>>> G = Graph(Integer(2))
>>> st(G, [[Integer(0),Integer(1)]], order=True)
([[1, 0]], Graph on 2 vertices, 2)
>>> st(G, [[Integer(0)],[Integer(1)]], order=True)
([], Graph on 2 vertices, 1)
>>> G.add_edge(Integer(0),Integer(1))
>>> st(G, [[Integer(0),Integer(1)]], order=True)
([[1, 0]], Graph on 2 vertices, 2)
>>> st(G, [[Integer(0)],[Integer(1)]], order=True)
([], Graph on 2 vertices, 1)
```

Graphs on three vertices:

```
sage: G = Graph(3)
sage: st(G, [[0,1,2]], order=True)
([[0, 2, 1], [1, 0, 2]], Graph on 3 vertices, 6)
sage: st(G, [[0],[1,2]], order=True)
([[0, 2, 1]], Graph on 3 vertices, 2)
sage: st(G, [[0],[1],[2]], order=True)
([], Graph on 3 vertices, 1)
sage: G.add_edge(0,1)
sage: st(G, [range(3)], order=True)
([[1, 0, 2]], Graph on 3 vertices, 2)
sage: st(G, [[0],[1,2]], order=True)
([], Graph on 3 vertices, 1)
sage: st(G, [[0,1],[2]], order=True)
([[1, 0, 2]], Graph on 3 vertices, 2)
```

```
>>> from sage.all import *
>>> G = Graph(Integer(3))
>>> st(G, [[Integer(0),Integer(1),Integer(2)]], order=True)
([[0, 2, 1], [1, 0, 2]], Graph on 3 vertices, 6)
>>> st(G, [[Integer(0)],[Integer(1),Integer(2)]], order=True)
([[0, 2, 1]], Graph on 3 vertices, 2)
>>> st(G, [[Integer(0)],[Integer(1)],[Integer(2)]], order=True)
([], Graph on 3 vertices, 1)
>>> G.add_edge(Integer(0),Integer(1))
>>> st(G, [range(Integer(3))], order=True)
([[1, 0, 2]], Graph on 3 vertices, 2)
>>> st(G, [[Integer(0)],[Integer(1),Integer(2)]], order=True)
([], Graph on 3 vertices, 1)
>>> st(G, [[Integer(0),Integer(1)],[Integer(2)]], order=True)
([[1, 0, 2]], Graph on 3 vertices, 2)
```

The Dodecahedron has automorphism group of size 120:

```
sage: G = graphs.DodecahedralGraph()
sage: Pi = [range(20)]
sage: st(G, Pi, order=True)[2]
120
```

```
>>> from sage.all import *
>>> G = graphs.DodecahedralGraph()
>>> Pi = [range(Integer(20))]
>>> st(G, Pi, order=True)[Integer(2)]
120
```

The three-cube has automorphism group of size 48:

```
sage: G = graphs.CubeGraph(3)
sage: G.relabel()
sage: Pi = [G.vertices(sort=False)]
sage: st(G, Pi, order=True)[2]
48
```

```
>>> from sage.all import *
>>> G = graphs.CubeGraph(Integer(3))
>>> G.relabel()
>>> Pi = [G.vertices(sort=False)]
>>> st(G, Pi, order=True)[Integer(2)]
48
```

We obtain the same output using different types of Sage graphs:

```
sage: G = graphs.DodecahedralGraph()
sage: GD = DenseGraph(20)
sage: GS = SparseGraph(20)
sage: for i,j,_ in G.edge_iterator():
....:  GD.add_arc(i,j); GD.add_arc(j,i)
....:  GS.add_arc(i,j); GS.add_arc(j,i)
sage: Pi = [range(20)]
sage: a,b = st(G, Pi)
sage: asp,bsp = st(GS, Pi)
sage: ade,bde = st(GD, Pi)
sage: bsg = Graph()
sage: bdg = Graph()
sage: for i in range(20):
....:     for j in range(20):
....:         if bsp.has_arc(i,j):
....:             bsg.add_edge(i,j)
....:         if bde.has_arc(i,j):
....:             bdg.add_edge(i,j)
sage: a, b.graph6_string()
([[0, 19, 3, 2, 6, 5, 4, 17, 18, 11, 10, 9, 13, 12, 16, 15, 14, 7, 8, 1],
  [0, 1, 8, 9, 13, 14, 7, 6, 2, 3, 19, 18, 17, 4, 5, 15, 16, 12, 11, 10],
  [1, 8, 9, 10, 11, 12, 13, 14, 7, 6, 2, 3, 4, 5, 15, 16, 17, 18, 19, 0]],
 'S?[PG__OQ@?_?_?P?CO?_?AE?EC?Ac?@O')
sage: a == asp
True
```

(continues on next page)

(continued from previous page)

```
sage: a == ade
True
sage: b == bsg
True
sage: b == bdg
True
```

```
>>> from sage.all import *
>>> G = graphs.DodecahedralGraph()
>>> GD = DenseGraph(Integer(20))
>>> GS = SparseGraph(Integer(20))
>>> for i,j,_ in G.edge_iterator():
...  GD.add_arc(i,j); GD.add_arc(j,i)
...  GS.add_arc(i,j); GS.add_arc(j,i)
>>> Pi = [range(Integer(20))]
>>> a,b = st(G, Pi)
>>> asp,bsp = st(GS, Pi)
>>> ade,bde = st(GD, Pi)
>>> bsg = Graph()
>>> bdg = Graph()
>>> for i in range(Integer(20)):
...     for j in range(Integer(20)):
...         if bsp.has_arc(i,j):
...             bsg.add_edge(i,j)
...         if bde.has_arc(i,j):
...             bdg.add_edge(i,j)
>>> a, b.graph6_string()
([[0, 19, 3, 2, 6, 5, 4, 17, 18, 11, 10, 9, 13, 12, 16, 15, 14, 7, 8, 1],
  [0, 1, 8, 9, 13, 14, 7, 6, 2, 3, 19, 18, 17, 4, 5, 15, 16, 12, 11, 10],
  [1, 8, 9, 10, 11, 12, 13, 14, 7, 6, 2, 3, 4, 5, 15, 16, 17, 18, 19, 0]],
 'S?[PG__OQ@?_?_?P?CO?_?AE?EC?Ac?@O')
>>> a == asp
True
>>> a == ade
True
>>> b == bsg
True
>>> b == bdg
True
```

Cubes!:

```
sage: C = graphs.CubeGraph(1)
sage: gens, order = st(C, [C.vertices(sort=False)], lab=False, order=True); order
2
sage: C = graphs.CubeGraph(2)
sage: gens, order = st(C, [C.vertices(sort=False)], lab=False, order=True); order
8
sage: C = graphs.CubeGraph(3)
sage: gens, order = st(C, [C.vertices(sort=False)], lab=False, order=True); order
48
sage: C = graphs.CubeGraph(4)
```

(continues on next page)

(continued from previous page)

```
sage: gens, order = st(C, [C.vertices(sort=False)], lab=False, order=True); order
384
sage: C = graphs.CubeGraph(5)
sage: gens, order = st(C, [C.vertices(sort=False)], lab=False, order=True); order
3840
sage: C = graphs.CubeGraph(6)
sage: gens, order = st(C, [C.vertices(sort=False)], lab=False, order=True); order
46080
```

```
>>> from sage.all import *
>>> C = graphs.CubeGraph(Integer(1))
>>> gens, order = st(C, [C.vertices(sort=False)], lab=False, order=True); order
2
>>> C = graphs.CubeGraph(Integer(2))
>>> gens, order = st(C, [C.vertices(sort=False)], lab=False, order=True); order
8
>>> C = graphs.CubeGraph(Integer(3))
>>> gens, order = st(C, [C.vertices(sort=False)], lab=False, order=True); order
48
>>> C = graphs.CubeGraph(Integer(4))
>>> gens, order = st(C, [C.vertices(sort=False)], lab=False, order=True); order
384
>>> C = graphs.CubeGraph(Integer(5))
>>> gens, order = st(C, [C.vertices(sort=False)], lab=False, order=True); order
3840
>>> C = graphs.CubeGraph(Integer(6))
>>> gens, order = st(C, [C.vertices(sort=False)], lab=False, order=True); order
46080
```

One can also turn off the indicator function (note: this will take longer):

```
sage: D1 = DiGraph({0:[2],2:[0],1:[1]}, loops=True)
sage: D2 = DiGraph({1:[2],2:[1],0:[0]}, loops=True)
sage: a,b = st(D1, [D1.vertices(sort=False)], dig=True, use_indicator_
↪function=False)
sage: c,d = st(D2, [D2.vertices(sort=False)], dig=True, use_indicator_
↪function=False)
sage: b==d
True
```

```
>>> from sage.all import *
>>> D1 = DiGraph({Integer(0):[Integer(2)],Integer(2):[Integer(0)],
↪Integer(1):[Integer(1)]}, loops=True)
>>> D2 = DiGraph({Integer(1):[Integer(2)],Integer(2):[Integer(1)],
↪Integer(0):[Integer(0)]}, loops=True)
>>> a,b = st(D1, [D1.vertices(sort=False)], dig=True, use_indicator_
↪function=False)
>>> c,d = st(D2, [D2.vertices(sort=False)], dig=True, use_indicator_
↪function=False)
>>> b==d
True
```

This example is due to Chris Godsil:

```
sage: HS = graphs.HoffmanSingletonGraph()
sage: alqs = [Set(c) for c in (HS.complement()).cliques_maximum()]
sage: Y = Graph([alqs, lambda s,t: len(s.intersection(t))==0])
sage: Y0,Y1 = Y.connected_components_subgraphs()
sage: st(Y0, [Y0.vertices(sort=False)])[1] == st(Y1, [Y1.vertices(sort=False)])[1]
True
sage: st(Y0, [Y0.vertices(sort=False)])[1] == st(HS, [HS.vertices(sort=False)])[1]
True
sage: st(HS, [HS.vertices(sort=False)])[1] == st(Y1, [Y1.vertices(sort=False)])[1]
True
```

```
>>> from sage.all import *
>>> HS = graphs.HoffmanSingletonGraph()
>>> alqs = [Set(c) for c in (HS.complement()).cliques_maximum()]
>>> Y = Graph([alqs, lambda s,t: len(s.intersection(t))==Integer(0)])
>>> Y0,Y1 = Y.connected_components_subgraphs()
>>> st(Y0, [Y0.vertices(sort=False)])[Integer(1)] == st(Y1, [Y1.
↪vertices(sort=False)])[Integer(1)]
True
>>> st(Y0, [Y0.vertices(sort=False)])[Integer(1)] == st(HS, [HS.
↪vertices(sort=False)])[Integer(1)]
True
>>> st(HS, [HS.vertices(sort=False)])[Integer(1)] == st(Y1, [Y1.
↪vertices(sort=False)])[Integer(1)]
True
```

Certain border cases need to be tested as well:

```
sage: G = Graph('Fll^G')
sage: a,b,c = st(G, [range(G.num_verts())], order=True); b
Graph on 7 vertices
sage: c
48
sage: G = Graph(21)
sage: st(G, [range(G.num_verts())], order=True)[2] == factorial(21)
True

sage: G = Graph('^????????????????????{??N??@w??FaGa?PCO@CP?AGa?_QO?Q@G?CcA??cc???
↪?Bo????{????F_')
sage: perm = {3:15, 15:3}
sage: H = G.relabel(perm, inplace=False)
sage: st(G, [range(G.num_verts())])[1] == st(H, [range(H.num_verts())])[1]
True

sage: st(Graph(':Dkw'), [range(5)], lab=False, dig=True)
[[4, 1, 2, 3, 0], [0, 2, 1, 3, 4]]
```

```
>>> from sage.all import *
>>> G = Graph('Fll^G')
>>> a,b,c = st(G, [range(G.num_verts())], order=True); b
Graph on 7 vertices
```

(continues on next page)

(continued from previous page)

```
>>> c
48
>>> G = Graph(Integer(21))
>>> st(G, [range(G.num_verts())], order=True)[Integer(2)] ==␣
↪factorial(Integer(21))
True

>>> G = Graph('^????????????????????{??N??@w??FaGa?PCO@CP?AGa?_QO?Q@G?CcA??cc????
↪Bo????{????F_')
>>> perm = {Integer(3):Integer(15), Integer(15):Integer(3)}
>>> H = G.relabel(perm, inplace=False)
>>> st(G, [range(G.num_verts())])[Integer(1)] == st(H, [range(H.num_
↪verts())])[Integer(1)]
True

>>> st(Graph(':Dkw'), [range(Integer(5))], lab=False, dig=True)
[[4, 1, 2, 3, 0], [0, 2, 1, 3, 4]]
```

30.4 Partition backtrack functions for lists – a simple example of using partn_ref

EXAMPLES:

```
sage: import sage.groups.perm_gps.partn_ref.refinement_lists
```

```
>>> from sage.all import *
>>> import sage.groups.perm_gps.partn_ref.refinement_lists
```

sage.groups.perm_gps.partn_ref.refinement_lists.**is_isomorphic**(*self*, *other*)

Return the bijection as a permutation if two lists are isomorphic, return `False` otherwise.

EXAMPLES:

```
sage: from sage.groups.perm_gps.partn_ref.refinement_lists import is_isomorphic
sage: is_isomorphic([0,0,1],[1,0,0])
[1, 2, 0]
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.partn_ref.refinement_lists import is_isomorphic
>>> is_isomorphic([Integer(0),Integer(0),Integer(1)],[Integer(1),Integer(0),
↪Integer(0)])
[1, 2, 0]
```

30.5 Partition backtrack functions for matrices

EXAMPLES:

```
sage: import sage.groups.perm_gps.partn_ref.refinement_matrices
```

```
>>> from sage.all import *
>>> import sage.groups.perm_gps.partn_ref.refinement_matrices
```

REFERENCE:

- [1] McKay, Brendan D. Practical Graph Isomorphism. Congressus Numerantium, Vol. 30 (1981), pp. 45-87.
- [2] Leon, Jeffrey. Permutation Group Algorithms Based on Partitions, I: Theory and Algorithms. J. Symbolic Computation, Vol. 12 (1991), pp. 533-583.

class sage.groups.perm_gps.partn_ref.refinement_matrices.**MatrixStruct**

Bases: object

automorphism_group()

Return a list of generators of the automorphism group, along with its order and a base for which the list of generators is a strong generating set.

For more examples, see self.run().

EXAMPLES:

```
sage: from sage.groups.perm_gps.partn_ref.refinement_matrices import␣
↪MatrixStruct

sage: M = MatrixStruct(matrix(GF(3),[[0,1,2],[0,2,1]]))
sage: M.automorphism_group()
([[0, 2, 1]], 2, [1])
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.partn_ref.refinement_matrices import␣
↪MatrixStruct

>>> M = MatrixStruct(matrix(GF(Integer(3)),[[Integer(0),Integer(1),
↪Integer(2)],[Integer(0),Integer(2),Integer(1)]]))
>>> M.automorphism_group()
([[0, 2, 1]], 2, [1])
```

canonical_relabeling()

Return a canonical relabeling (in list permutation format).

For more examples, see self.run().

EXAMPLES:

```
sage: from sage.groups.perm_gps.partn_ref.refinement_matrices import␣
↪MatrixStruct

sage: M = MatrixStruct(matrix(GF(3),[[0,1,2],[0,2,1]]))
sage: M.canonical_relabeling()
[0, 1, 2]
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.partn_ref.refinement_matrices import␣
↪MatrixStruct

>>> M = MatrixStruct(matrix(GF(Integer(3)),[[Integer(0),Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2)],[Integer(0),Integer(2),Integer(1)]]))
>>> M.canonical_relabeling()
[0, 1, 2]
```

display()

Display the matrix, and associated data.

EXAMPLES:

```
sage: from sage.groups.perm_gps.partn_ref.refinement_matrices import␣
↪MatrixStruct
sage: M = MatrixStruct(Matrix(GF(5), [[0,1,1,4,4],[0,4,4,1,1]]))
sage: M.display()
[0 1 1 4 4]
[0 4 4 1 1]

01100
00011
1

00011
01100
4
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.partn_ref.refinement_matrices import␣
↪MatrixStruct
>>> M = MatrixStruct(Matrix(GF(Integer(5)), [[Integer(0),Integer(1),
↪Integer(1),Integer(4),Integer(4)],[Integer(0),Integer(4),Integer(4),
↪Integer(1),Integer(1)]]))
>>> M.display()
[0 1 1 4 4]
[0 4 4 1 1]
<BLANKLINE>
01100
00011
1
<BLANKLINE>
00011
01100
4
```

is_isomorphic(*other*)

Calculate whether `self` is isomorphic to `other`.

EXAMPLES:

```
sage: from sage.groups.perm_gps.partn_ref.refinement_matrices import␣
↪MatrixStruct
sage: M = MatrixStruct(Matrix(GF(11), [[1,2,3,0,0,0],[0,0,0,1,2,3]]))
sage: N = MatrixStruct(Matrix(GF(11), [[0,1,0,2,0,3],[1,0,2,0,3,0]]))
sage: M.is_isomorphic(N)
[0, 2, 4, 1, 3, 5]
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.partn_ref.refinement_matrices import␣
↪MatrixStruct
>>> M = MatrixStruct(Matrix(GF(Integer(11)), [[Integer(1),Integer(2),
↪Integer(3),Integer(0),Integer(0),Integer(0)],[Integer(0),Integer(0),
↪Integer(0),Integer(1),Integer(2),Integer(3)]]))
>>> N = MatrixStruct(Matrix(GF(Integer(11)), [[Integer(0),Integer(1),
↪Integer(0),Integer(2),Integer(0),Integer(3)],[Integer(1),Integer(0),
↪Integer(2),Integer(0),Integer(3),Integer(0)]]))
>>> M.is_isomorphic(N)
[0, 2, 4, 1, 3, 5]
```

run(*partition=None*)

Perform the canonical labeling and automorphism group computation, storing results to `self`.

INPUT:

- `partition` – an optional list of lists partition of the columns; default is the unit partition

EXAMPLES:

```
sage: from sage.groups.perm_gps.partn_ref.refinement_matrices import␣
↪MatrixStruct

sage: M = MatrixStruct(matrix(GF(3),[[0,1,2],[0,2,1]]))
sage: M.run()
sage: M.automorphism_group()
([[0, 2, 1]], 2, [1])
sage: M.canonical_relabeling()
[0, 1, 2]

sage: M = MatrixStruct(matrix(GF(3),[[0,1,2],[0,2,1],[1,0,2],[1,2,0],[2,0,1],
↪[2,1,0]]))
sage: M.automorphism_group()[1] == 6
True

sage: M = MatrixStruct(matrix(GF(3),[[0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,2]]))
sage: M.automorphism_group()[1] == factorial(14)
True
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.partn_ref.refinement_matrices import␣
↪MatrixStruct

>>> M = MatrixStruct(matrix(GF(Integer(3)),[[Integer(0),Integer(1),
↪Integer(2)],[Integer(0),Integer(2),Integer(1)]]))
>>> M.run()
>>> M.automorphism_group()
([[0, 2, 1]], 2, [1])
>>> M.canonical_relabeling()
[0, 1, 2]

>>> M = MatrixStruct(matrix(GF(Integer(3)),[[Integer(0),Integer(1),
↪Integer(2)],[Integer(0),Integer(2),Integer(1)],[Integer(1),Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2)],[Integer(1),Integer(2),Integer(0)],[Integer(2),Integer(0),
↪Integer(1)],[Integer(2),Integer(1),Integer(0)]]))
>>> M.automorphism_group()[Integer(1)] == Integer(6)
True

>>> M = MatrixStruct(matrix(GF(Integer(3)),[[Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(1),
↪Integer(2)]]))
>>> M.automorphism_group()[Integer(1)] == factorial(Integer(14))
True
```

sage.groups.perm_gps.partn_ref.refinement_matrices.**random_tests**(*n=10*, *nrows_max=50*, *ncols_max=50*, *nsymbols_max=10*, *perms_per_matrix=5*, *density_range=(0.1, 0.9)*)

Test to make sure that `C(gamma(M)) == C(M)` for random permutations `gamma` and random matrices `M`, and that `M.is_isomorphic(gamma(M))` returns an isomorphism.

INPUT:

- `n` – run tests on this many matrices
- `nrows_max` – test matrices with at most this many rows
- `ncols_max` – test matrices with at most this many columns
- `perms_per_matrix` – test each matrix with this many random permutations
- `nsymbols_max` – maximum number of distinct symbols in the matrix

This code generates n random matrices M on at most ncols_max columns and at most nrows_max rows. The density of entries in the basis is chosen randomly between 0 and 1.

For each matrix M generated, we uniformly generate perms_per_matrix random permutations and verify that the canonical labels of M and the image of M under the generated permutation are equal, and that the isomorphism is discovered by the double coset function.

CHAPTER

THIRTYONE

INTERNALS

31.1 Base for Classical Matrix Groups

This module implements the base class for matrix groups that have various famous names, like the general linear group.

EXAMPLES:

```
sage: SL(2, ZZ)
Special Linear Group of degree 2 over Integer Ring
sage: G = SL(2, GF(3)); G
Special Linear Group of degree 2 over Finite Field of size 3

sage: # needs sage.libs.gap
sage: G.is_finite()
True
sage: G.conjugacy_classes_representatives()
(
[1 0]  [0 2]  [0 1]  [2 0]  [0 2]  [0 1]  [0 2]
[0 1], [1 1], [2 1], [0 2], [1 2], [2 2], [1 0]
)
sage: G = SL(6, GF(5))
sage: G.gens()
(
[2 0 0 0 0 0]  [4 0 0 0 0 1]
[0 3 0 0 0 0]  [4 0 0 0 0 0]
[0 0 1 0 0 0]  [0 4 0 0 0 0]
[0 0 0 1 0 0]  [0 0 4 0 0 0]
[0 0 0 0 1 0]  [0 0 0 4 0 0]
[0 0 0 0 0 1], [0 0 0 0 4 0]
)
```

```
>>> from sage.all import *
>>> SL(Integer(2), ZZ)
Special Linear Group of degree 2 over Integer Ring
>>> G = SL(Integer(2), GF(Integer(3))); G
Special Linear Group of degree 2 over Finite Field of size 3

>>> # needs sage.libs.gap
>>> G.is_finite()
True
>>> G.conjugacy_classes_representatives()
```

(continues on next page)

(continued from previous page)

```
(
[1 0]   [0 2]   [0 1]   [2 0]   [0 2]   [0 1]   [0 2]
[0 1], [1 1], [2 1], [0 2], [1 2], [2 2], [1 0]
)
>>> G = SL(Integer(6), GF(Integer(5)))
>>> G.gens()
(
[2 0 0 0 0 0]  [4 0 0 0 0 1]
[0 3 0 0 0 0]  [4 0 0 0 0 0]
[0 0 1 0 0 0]  [0 4 0 0 0 0]
[0 0 0 1 0 0]  [0 0 4 0 0 0]
[0 0 0 0 1 0]  [0 0 0 4 0 0]
[0 0 0 0 0 1], [0 0 0 0 4 0]
)
```

class sage.groups.matrix_gps.named_group.**NamedMatrixGroup_generic**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *category=None*, *invariant_form=None*)

Bases: `CachedRepresentation`, *MatrixGroup_generic*

Base class for "named" matrix groups.

INPUT:

- `degree` – integer; the degree (number of rows/columns of matrices)
- `base_ring` – ring; the base ring of the matrices
- `special` – boolean; whether the matrix group is special, that is, elements have determinant one
- `sage_name` – string; the name of the group
- `latex_string` – string; the latex representation
- `category` – (optional) a subcategory of `sage.categories.groups.Groups` passed to the constructor of *sage.groups.matrix_gps.matrix_group.MatrixGroup_generic*
- `invariant_form` – (optional) square-matrix of the given degree over the given base_ring describing a bilinear form to be kept invariant by the group

EXAMPLES:

```
sage: G = GL(2, QQ)
sage: from sage.groups.matrix_gps.named_group import NamedMatrixGroup_generic
sage: isinstance(G, NamedMatrixGroup_generic)
True
```

```
>>> from sage.all import *
>>> G = GL(Integer(2), QQ)
>>> from sage.groups.matrix_gps.named_group import NamedMatrixGroup_generic
>>> isinstance(G, NamedMatrixGroup_generic)
True
```

> **See also**
>
> See the examples for `GU()`, `SU()`, `Sp()`, etc. as well.

`sage.groups.matrix_gps.named_group.`**`normalize_args_invariant_form`**`(`*R*, *d*, *invariant_form*`)`

Normalize the input of a user defined invariant bilinear form for orthogonal, unitary and symplectic groups.

Further informations and examples can be found in the defining functions (`GU()`, `SU()`, `Sp()`, etc.) for unitary, symplectic groups, etc.

INPUT:

- `R` – instance of the integral domain which should become the `base_ring` of the classical group
- `d` – integer giving the dimension of the module the classical group is operating on
- `invariant_form` – (optional) instances being accepted by the matrix-constructor that define a $d \times d$ square matrix over R describing the bilinear form to be kept invariant by the classical group

OUTPUT:

`None` if `invariant_form` was not specified (or `None`). A matrix if the normalization was possible; otherwise an error is raised.

AUTHORS:

- Sebastian Oehms (2018-8) (see Issue #26028)

`sage.groups.matrix_gps.named_group.`**`normalize_args_vectorspace`**`(`**args*, ***kwds*`)`

Normalize the arguments that relate to a vector space.

INPUT:

Something that defines an affine space. For example

- An affine space itself:
  - `A` – affine space
- A vector space:
  - `V` – a vector space
- Degree and base ring:
  - `degree` – integer; the degree of the affine group, that is, the dimension of the affine space the group is acting on
  - `ring` – a ring or an integer. The base ring of the affine space. If an integer is given, it must be a prime power and the corresponding finite field is constructed.
  - `var='a'` – (optional) keyword argument to specify the finite field generator name in the case where `ring` is a prime power

OUTPUT: a pair `(degree, ring)`

31.2 Base for Classical Matrix Groups with GAP

`class` `sage.groups.matrix_gps.named_group_gap.`**`NamedMatrixGroup_gap`**`(`*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *gap_command_string*, *category=None*`)`

Bases: *NamedMatrixGroup_generic*, *MatrixGroup_gap*

Base class for "named" matrix groups using LibGAP.

INPUT:

- `degree` – integer; the degree (number of rows/columns of matrices)
- `base_ring` – ring; the base ring of the matrices
- `special` – boolean; whether the matrix group is special, that is, elements have determinant one
- `latex_string` – string; the latex representation
- `gap_command_string` – string; the GAP command to construct the matrix group

EXAMPLES:

```
sage: G = GL(2, GF(3))
sage: from sage.groups.matrix_gps.named_group_gap import NamedMatrixGroup_gap
sage: isinstance(G, NamedMatrixGroup_gap)
True
```

```
>>> from sage.all import *
>>> G = GL(Integer(2), GF(Integer(3)))
>>> from sage.groups.matrix_gps.named_group_gap import NamedMatrixGroup_gap
>>> isinstance(G, NamedMatrixGroup_gap)
True
```

CHAPTER

THIRTYTWO

INDICES AND TABLES

- Index
- Module Index
- Search Page

BIBLIOGRAPHY

[Cohen1] H. Cohen, Advanced topics in computational number theory, Springer, 2000.

[Cohen2] H. Cohen, A course in computational algebraic number theory, Springer, 1996.

[Rotman] J. Rotman, An introduction to the theory of groups, 4th ed, Springer, 1995.

PYTHON MODULE INDEX

g

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y