Sage 9.5 Reference Manual: Groups

Release 9.5

The Sage Development Team

Jan 31, 2022
<table>
<thead>
<tr>
<th></th>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Examples of Groups</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Base class for groups</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Group homomorphisms for groups with a GAP backend</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>LibGAP-based Groups</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>Generic LibGAP-based Group</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>Mix-in Class for GAP-based Groups</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>PARI Groups</td>
<td>35</td>
</tr>
<tr>
<td>8</td>
<td>Miscellaneous generic functions</td>
<td>37</td>
</tr>
<tr>
<td>9</td>
<td>Free Groups</td>
<td>51</td>
</tr>
<tr>
<td>10</td>
<td>Finitely Presented Groups</td>
<td>59</td>
</tr>
<tr>
<td>11</td>
<td>Named Finitely Presented Groups</td>
<td>79</td>
</tr>
<tr>
<td>12</td>
<td>Braid groups</td>
<td>85</td>
</tr>
<tr>
<td>13</td>
<td>Cubic Braid Groups</td>
<td>111</td>
</tr>
<tr>
<td>14</td>
<td>Indexed Free Groups</td>
<td>125</td>
</tr>
<tr>
<td>15</td>
<td>Right-Angled Artin Groups</td>
<td>129</td>
</tr>
<tr>
<td>16</td>
<td>Functor that converts a commutative additive group into a multiplicative group.</td>
<td>135</td>
</tr>
<tr>
<td>17</td>
<td>Semidirect product of groups</td>
<td>139</td>
</tr>
<tr>
<td>18</td>
<td>Miscellaneous Groups</td>
<td>145</td>
</tr>
<tr>
<td>19</td>
<td>Semimonomial transformation group</td>
<td>147</td>
</tr>
<tr>
<td>20</td>
<td>Elements of a semimonomial transformation group</td>
<td>151</td>
</tr>
<tr>
<td>21</td>
<td>Class functions of groups</td>
<td>155</td>
</tr>
<tr>
<td>22</td>
<td>Conjugacy classes of groups</td>
<td>167</td>
</tr>
</tbody>
</table>
### 23 Abelian Groups

23.1 Multiplicative Abelian Groups ................................................................. 171
23.2 Finitely generated abelian groups with GAP ........................................ 187
23.3 Automorphisms of abelian groups ............................................................ 195
23.4 Multiplicative Abelian Groups With Values ........................................... 200
23.5 Dual groups of Finite Multiplicative Abelian Groups ............................ 205
23.6 Base class for abelian group elements ..................................................... 208
23.7 Abelian group elements ............................................................................ 211
23.8 Elements (characters) of the dual group of a finite Abelian group .......... 213
23.9 Homomorphisms of abelian groups .......................................................... 215
23.10 Additive Abelian Groups ......................................................................... 217
23.11 Wrapper class for abelian groups ............................................................. 221
23.12 Groups of elements representing (complex) arguments .......................... 223
23.13 Groups of imaginary elements ................................................................. 229

### 24 Permutation Groups

24.1 Catalog of permutation groups ................................................................. 233
24.2 Constructor for permutations .................................................................... 233
24.3 Permutation groups .................................................................................... 236
24.4 “Named” Permutation groups (such as the symmetric group, S_n) ......... 288
24.5 Permutation group elements ..................................................................... 319
24.6 Permutation group homomorphisms .......................................................... 328
24.7 Rubik’s cube group functions ...................................................................... 331
24.8 Conjugacy Classes Of The Symmetric Group ......................................... 345

### 25 Matrix and Affine Groups

25.1 Library of Interesting Groups .................................................................... 349
25.2 Base classes for Matrix Groups .................................................................. 349
25.3 Matrix Group Elements ............................................................................. 354
25.4 Finitely Generated Matrix Groups ............................................................. 360
25.5 Homomorphisms Between Matrix Groups ............................................... 373
25.6 Matrix Group Homsets .............................................................................. 373
25.7 Binary Dihedral Groups ............................................................................. 374
25.8 Coxeter Groups As Matrix Groups ............................................................ 375
25.9 Linear Groups ............................................................................................ 383
25.10 Orthogonal Linear Groups ....................................................................... 386
25.11 Groups of isometries .................................................................................. 395
25.12 Symplectic Linear Groups ........................................................................ 397
25.13 Unitary Groups GU(n, q) and SU(n, q) ..................................................... 401
25.14 Heisenberg Group .................................................................................... 406
25.15 Affine Groups ........................................................................................... 407
25.16 Euclidean Groups ..................................................................................... 412
25.17 Elements of Affine Groups ....................................................................... 415

### 26 Lie Groups

26.1 Nilpotent Lie groups .................................................................................. 419

### 27 Partition Refinement

27.1 Canonical augmentation ............................................................................ 431
27.2 Data structures ........................................................................................... 431
27.3 Graph-theoretic partition backtrack functions .......................................... 434
27.4 Partition backtrack functions for lists – a simple example of using partn_ref 441
27.5 Partition backtrack functions for matrices .................................................. 442

### 28 Internals

445
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.1 Base for Classical Matrix Groups</td>
<td>445</td>
</tr>
<tr>
<td>29 Indices and Tables</td>
<td>449</td>
</tr>
<tr>
<td>Bibliography</td>
<td>451</td>
</tr>
<tr>
<td>Python Module Index</td>
<td>453</td>
</tr>
<tr>
<td>Index</td>
<td>455</td>
</tr>
</tbody>
</table>
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 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`
• Finitely Presented Groups (groups.presentation.<tab>)
  - groups.presentation.Alternating
  - 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.CoxeterGroup
    * groups.misc.ReflectionGroup
    * groups.misc.RightAngledArtin
    * groups.misc.WeylGroup
  - other miscellaneous groups
    * groups.misc.AdditiveAbelian
    * groups.misc.AdditiveCyclic
    * groups.misc.Free
    * groups.misc.SemimonomialTransformation
class sage.groups.group.AbelianGroup
    Bases: sage.groups.group.Group
    
    Generic abelian group.

    is_abelian()
    Return True.

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

class sage.groups.group.AlgebraicGroup
    Bases: sage.groups.group.Group

class sage.groups.group.FiniteGroup
    Bases: sage.groups.group.Group
    
    Generic finite group.

    is_finite()
    Return True.

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

class sage.groups.group.Group
    Bases: sage.structure.parent.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

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()
False
```

is_finite()
Returns 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):
...
NotImplementedError
```

is_multiplicative()
Returns 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
```

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
```

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()
Traceback (most recent call last):
  ... Not ImplementedError

sage.groups.group.is_Group(x)

Return whether x is a group object.

INPUT:

• x – anything.

OUTPUT:

Boolean.

EXAMPLES:

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

sage: from sage.groups.group import is_Group
sage: is_Group(F)
True
sage: is_Group("a string")
False
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>)
```

AUTHORS:

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

```
class sage.groups.libgap_morphism.GroupHomset_libgap(G, H, category=None, check=True)
    Bases: sage.categories.homset.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
```
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. Else 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)
```

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

Bases: sage.categories.morphism.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 – (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)
```

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)
```

(continues on next page)
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 >

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)

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 ]

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: 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)

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

Return the preimage of the subgroup $S$.

INPUT:

• $S$ – a subgroup of this group

EXAMPLES:

```python
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]
[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:

```python
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: G = GU(3,2)
```

(continues on next page)
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)

section()
This method returns 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: 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]
This module provides helper class for wrapping GAP groups via \texttt{libgap}. See \texttt{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 \texttt{ParentLibGAP}

\begin{verbatim}
sage: from sage.groups.libgap_wrapper import ElementLibGAP, ParentLibGAP
element = FooGroup().an_element()
element
f1
\end{verbatim}

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

\begin{verbatim}
sage: FooGroup()
<pc group of size 3 with 1 generators>
sage: type(FooGroup().gap())
<class 'sage.libs.gap.element.GapElement'>
\end{verbatim}

The element class is a subclass of \texttt{MultiplicativeGroupElement}. To use it, you just inherit from \texttt{ElementLibGAP}

\begin{verbatim}
sage: element = FooGroup().an_element()
sage: element
f1
\end{verbatim}

The element class implements group operations and printing via LibGAP:

\begin{verbatim}
sage: element._repr_()
'f1'
sage: element * element
f1^2
\end{verbatim}

AUTHORS:
- Volker Braun
class sage.groups.libgap_wrapper.ElementLibGAP
Bases: sage.structure.element.MultiplicativeGroupElement

A class for LibGAP-based Sage group elements

INPUT:

• parent – the Sage parent
• libgap_element – the libgap element that is being wrapped

EXAMPLES:

```python
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 generators>
sage: FooGroup().gens()
(f1,)
```

gap()

Return a LibGAP representation of the element.

OUTPUT:

A GapElement

EXAMPLES:

```python
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'>
```

inverse()

Return the inverse of self.

is_conjugate(other)

Return whether the elements self and other are conjugate.

EXAMPLES:

```python
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)
```

(continues on next page)
False
sage: a.is_conjugate((a*b^2) * a * ~(a*b^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

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
sage: z = Mod(0, 3)
sage: o = Mod(1, 3)
sage: 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

nth_roots(n)
Return the set of n-th roots of this group element.
EXAMPLES:

```python
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*Z(3) ], [ 0*Z(3), Z(3), 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
```

order()  
Return the multiplicative order.

EXAMPLES:

```python
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
```

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

Bases: sage.structure.sage_object.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:

```python
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 generators>

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

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'>

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 ]

We can also convert directly to libgap:

sage: libgap(GL(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 \(\text{ngens}()\) (exclusive). The index of the generator.

**OUTPUT:**
The \(i\)-th generator of the group.

**EXAMPLES:**

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

**generators**()
Return the generators of the group.

**EXAMPLES:**

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

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

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

**gens**()
Return the generators of the group.

**EXAMPLES:**

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

**generators**() is an alias for **gens**()
```python
sage: G = FreeGroup('a, b')
sage: G.generators()
(a, b)
sage: H = FreeGroup(3, 'x')
sage: 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:**
```python
sage: G = FreeGroup(3)
sage: G.is_subgroup()
False
```

**ngens()**
Return the number of generators of self.

**OUTPUT:**
Integer.

**EXAMPLES:**
```python
sage: G = FreeGroup(2)
sage: G.ngens()
2
```

**one()**
Return the identity element of self.

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

**subgroup(generators)**
Return the subgroup generated.

**INPUT:**
- `generators` – a list/tuple/iterable of group elements.

**OUTPUT:**
The subgroup generated by `generators`.

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

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

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

class sage.groups.libgap_group.GroupLibGAP(*args, **kwds):
    Bases: sage.groups.libgap_mixin.GroupMixinLibGAP, sage.groups.group.Group, sage.groups.libgap_wrapper.ParentLibGAP
    Group interface for LibGAP-based groups.
    INPUT:
    Same as ParentLibGAP.
    Element
    alias of sage.groups.libgap_wrapper.ElementLibGAP
```
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`.

```python
class sage.groups.libgap_mixin.GroupMixinLibGAP
    Bases: object

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

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

center()
Return the center of this linear group as a subgroup.

OUTPUT:
The center as a subgroup.

EXAMPLES:

sage: G = SU(3,GF(2))
sage: G.center()
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()
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] 
)
**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* – a 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()))
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):
  ... Not ImplementedError: 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()
[ [ 1 -1]
  [ 1 1]

Sage: MatrixGroup(SymmetricGroup(3)).character_table()
[ [ 1 1 -1]
  [ 2 -1 0]
  [ 1 1 1]

Sage: MatrixGroup(SymmetricGroup(5)).character_table()
[ [ 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:**
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 $\{x^{-1}gx | x \in G\}$.

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
```

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)
```

```
sage: GL(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))
sage: len(G.conjugacy_classes_representatives())
16
```

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

(continues on next page)
```python
sage: len(GU(2,GF(5)).conjugacy_classes_representatives())
36
```

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

Traceback (most recent call last):
...

```
NotImplementedError: only implemented for finite groups
```

**intersection** *(other)*

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

**EXAMPLES:**

```python
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: 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] [ 0 0 1], [2 0 0] [-2 -1 2] [ 0 0 1])
```

```python
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

```python
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] [ 0 0 1], [2 0 0] [-2 -1 2] [ 0 0 1])

**irreducible_characters()**

Return the irreducible characters of the group.

**OUTPUT:**

A tuple containing all irreducible characters.

**EXAMPLES:**

```python
sage: G = GL(2,2)
sage: G.irreducible_characters()
```

(continues on next page)
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
```

**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
```

**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
```

**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)

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

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

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

is_polycyclic()
Return whether this group is polycyclic.

EXAMPLES:
is_simple()
Return whether this group is simple.

EXAMPLES:

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

is_solvable()
Return whether this group is solvable.

EXAMPLES:

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

is_supersolvable()
Return whether this group is supersolvable.

EXAMPLES:

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

list()
List all elements of this group.

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

EXAMPLES:

```python
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

An example over a ring (see trac ticket #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])

An example over a field (see trac ticket #10515):

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

Another example over a ring (see trac ticket #9437):

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

An error is raised if the group is not finite:

sage: GL(2,ZZ).list()
Traceback (most recent call last):
... Not ImplementedError: group must be finite
order()

Implements EnumeratedSets.ParentMethods.cardinality().

EXAMPLES:

```sage
g = Sp(4,GF(3))
g.order()
51840

g = SL(4,GF(3))
g.order()
12130560

g = GF(5); MS = MatrixSpace(F,2,2)
g = MatrixGroup(gens)
g.order()
480
```

random_element()

Return a random element of this group.

OUTPUT:

A group element.

EXAMPLES:

```sage
g = Sp(4,GF(3))
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

trivial_character()

Return the trivial character of this group.

OUTPUT: a group character

EXAMPLES:

sage: MatrixGroup(SymmetricGroup(3)).trivial_character()
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
PARI GROUPS

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

```python
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
```

cardinality()
Return the order of self.

```python
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
```

degree()
Return the degree of this group.

```python
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
```

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

```python
EXAMPLES:

sage: R.<x> = QQ[]
sage: f1 = x^4 - 17*x^3 - 2*x + 1
```
order()
Return the order of self.

EXAMPLES:

```sage
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
```

permutation_group()
Return the corresponding GAP transitive group

EXAMPLES:

```sage
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
```

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

EXAMPLES:

```sage
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
```

transitive_number()
If the transitive label is nTk, return k.

EXAMPLES:

```sage
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
```
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.

EXEMPLARY:

Some examples in the multiplicative group of a finite field:

• Discrete logs:

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

• Linear relation finder:

  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

• Orders of elements:

  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='*')
(continues on next page)
Some examples in the group of points of an elliptic curve over a finite field:

- **Discrete logs:**

  ```python
  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
  ```

- **Linear relation finder:**

  ```python
  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
  ```

- **Orders of elements:**

  ```python
  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
  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
  ```

`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 returning inverse of x
- op() - function of 2 arguments x, y returning x*y in group

OUTPUT:

An integer n such that an = 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
text
```

An additive example in an elliptic curve group:

```sage
text
```

This will return a multiple of the order of P:
sage: bsgs(P,P.parent()(0),Hasse_bounds(F.order()),operation='+')  
69327408

AUTHOR:  
• John Cremona (2008-03-15)

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

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 group

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: 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 and Baby step giant step.

EXAMPLES:

sage: b = Mod(2,37); a = b^20  
sage: discrete_log(a, b)  
20  
sage: b = Mod(2,997); a = b^20  
sage: discrete_log(a, b)  
20  
  
sage: K = GF(3^6,'b')  
sage: b = K.gen()  
sage: a = b^210  
sage: discrete_log(a, b, K.order()-1)  
210

(continues on next page)
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

See trac ticket #2356:

sage: F.<w> = GF(121)
sage: v = w^120
sage: v.log(w)
0
sage: K.<z>=CyclotomicField(230)
sage: w=z^50
sage: discrete_log(w,z)
50

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

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

In this case we cannot detect negative powers:

sage: eta = eps^(-3)
sage: discrete_log(eta,eps,bounds=(0,100))
Traceback (most recent call last):
  ...  
ValueError: No discrete log of -11515*a - 55224 found to base 5*a - 24

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

sage: - discrete_log(eta^-1,eps,bounds=(0,100))
-3

An additive example: elliptic curve DLOG:

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

An example of big smooth group:

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

AUTHORS:

• William Stein and David Joyner (2005-01-05)
• John Cremona (2008-02-29) rewrite using dict() and make generic

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

Alias for discrete_log.

sage.groups.generic.discrete_log_lambda(a, base, bounds, operation='*', 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’
• hash_function – having an efficient hash function is critical for this algorithm

OUTPUT: Returns an 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)
sage: discrete_log_lambda(a^1234567, a, (1200000, 1250000))
1234567

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)
This will return a multiple of the order of P:

```python
sage: discrete_log_lambda(P.parent()(0), P, Hasse_bounds(F.order()), operation='+')
69327408
```

```python
sage: K.<a> = GF(89**5)
```

```python
sage: hs = lambda x: hash(x) + 15
```

```python
sage: discrete_log_lambda(a**(89**3 - 3), a, (89**2, 89**4), operation='*', hash_function=hs)  # long time (10s on sage.math, 2011)
704966
```

AUTHOR:

– Yann Laigle-Chapuy (2009-01-25)

`sage.groups.generic.discrete_log_rho(a, base, ord=None, operation='*', 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` – a string (default: `'*'`) denoting whether we are in an additive group or a multiplicative one
• `hash_function` – having an efficient hash function is critical for this algorithm (see examples)

OUTPUT: an integer \( n \) such that \( a = base^n \) (or \( a = n \times base \))

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

EXAMPLES:

```python
sage: F.<a> = GF(2^13)
sage: g = F.gen()
sage: discrete_log_rho(g^1234, g)
1234
```

```python
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
```

It also works with matrices:

```python
sage: A = matrix(GF(50021),[[10577,23999,28893],[14601,41019,30188],[3081,736,27092]])
sage: discrete_log_rho(A^1234567, A)
1234567
```

Beware, the order must be prime:
If it fails to find a suitable logarithm, it raises a `ValueError`:

```python
sage: I = IntegerModRing(171980)
sage: discrete_log_rho(I(31002),I(15501))
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 $\text{hash}(x \ast y) = \text{hash}(x) + \text{hash}(y)$:

```python
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
```

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

```python
sage: discrete_log_rho(I(123456),I(1),operation='+', hash_function=lambda x: ~hash(x*x))
123456
```

AUTHOR:

• Yann Laigle-Chapuy (2009-09-05)

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

Function which solves the equation $aP = mQ$ 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):

```python
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,'+')
```
A multiplicative example (in a finite field’s multiplicative group):

```python
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
```

```
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 \(n_3 = \text{lcm}(n_1, n_2)\).

**EXAMPLES:**

```python
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: 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()
45
```
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

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)
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

Idempotence is detected, making the following fast:

sage: multiple(1,10^1000)
1

sage: E=EllipticCurve('389a1')
sage: P=E(-1,1)
sage: multiple(P,10,'+')
(645656132358737542773209599489/22817025904944891235367494656 : 5255321761242192881231818644174845702936831/
3446581505217248068297884384990762467229696 : 1)
sage: multiple(P,-10,'+')
(645656132358737542773209599489/22817025904944891235367494656 :
52897875762949844949529783029165608170166527/
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')
sage: P=E(-1,1)
sage: for Q in multiples(P,5): print((Q, Q.height()/P.height()))
((0 : 1 : 0), 0.000000000000000)
((-1 : 1 : 1), 1.000000000000000)
((10/9 : -35/27 : 1), 4.000000000000000)
((26/361 : -5720/6859 : 1), 9.000000000000000)
((47503/16641 : 9862190/2146689 : 1), 16.000000000000000)
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

next()
Return the next item in this multiples iterator.

next()
• P - a Sage object which is a group element
• bounds - a 2-tuple (lb, ub) such that \( m \cdot P = 0 \) (or \( P^m = 1 \)) for some \( m \) with \( lb \leq m \leq ub \).
• 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:**

```python
sage: from sage.groups.generic import order_from_bounds
sage: k.<a> = GF(5^5)
sage: b = a^4
sage: order_from_bounds(b,(5^4,5^5),operation='*')
781
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: K.<z>=CyclotomicField(230)
sage: w=z^50
sage: order_from_bounds(w,(200,250),operation='*')
23
```

**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:**

```python
sage: from sage.groups.generic import order_from_multiple
sage: order_from_multiple(P, m, plist=None, factorization=None, check=True, operation='+)
```

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

**INPUT:**

• P - a Sage object which is a group element;
• m - a Sage integer which is a multiple of the order of P, i.e. we require that \( m \cdot P = 0 \) (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 - a list of the prime factors of \( m \), or None - kept for compatibility only, prefer the use of factorization;
• operation - string: ‘+’ (default) or ‘*’.

**Note:** It is more efficient for the caller to factor \( m \) and cache the factors for subsequent calls.
EXAMPLES:

```python
sage: from sage.groups.generic import order_from_multiple
sage: k.<a> = GF(5^5)
sage: b = a^4
sage: order_from_multiple(b, 5^5-1, operation='*')
781
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: K.<z>=CyclotomicField(230)
sage: w=z^50
sage: order_from_multiple(w, 230, operation='*')
23
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: K.<a> = GF(3^60)
sage: order_from_multiple(a, 3^60-1, operation='*', check=False)
423911582752162013514294443200
```

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

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

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

```python
sage: D3 = DihedralGroup(3)
sage: D3.structure_description()
'S3'
```

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

```python
sage: D4 = DihedralGroup(4)
sage: D4.structure_description()
'D4'
```

Works for finitely presented groups (trac ticket #17573):

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

And matrix groups (trac ticket #17573):

```python
sage: groups.matrix.GL(4,2).structure_description()
'A8'
```
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
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}
```

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
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
```

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

```sage
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]
```

AUTHORS:
Construct a Free Group.

**INPUT:**

- `n` – integer or `None` (default). 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` – (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:**

```python
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 {}
```

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'`

```python
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}
```

We give two examples using the `index_set` option:

```python
sage: FreeGroup(index_set=ZZ)
Free group indexed by Integer Ring
sage: FreeGroup(index_set=ZZ, abelian=True)
Free abelian group indexed by Integer Ring
```

class `sage.groups.free_group.FreeGroupElement`

**Bases:** `sage.groups.libgap_wrapper.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`. 

52 Chapter 9. Free Groups
EXAMPLES:

```python
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, 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^-1
sage: x^(-1)
(b*a*b^-1*a^-1)
sage: x == x*y*y^(-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:**

A tuple of integers.

**EXAMPLES:**

```python
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)
```

**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

\[
\partial_j(x_i) = \delta_{i,j} \quad \text{for all indices } i \text{ and } j
\]

and

\[
\partial_j(uv) = \partial_j(u) + a(u)\partial_j(v) \quad \text{for all } u, v \in F.
\]

This map \( \partial_j \) 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 \( \mathbb{Z}[F] \) of \( F \) over \( \mathbb{Z} \), and when \( a_i = x_i \in A \). In this case, \( \partial_j \) 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 $\partial_j$.

• 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 $\mathbb{Z}[F]$.

• 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)
-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
```

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]),[1,2,3])
x1 - x1*x0^-1*x2^2*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])
t - t^2*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)
```

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
```

(continues on next page)
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 non-zero integer.

EXAMPLES:

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

class `sage.groups.free_group.FreeGroup_class`(generator_names, libgap_free_group=None)

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 \( i_1 \ldots i_j \), such that the abelianization of the group is isomorphic to

\[\mathbb{Z}/(i_1) \times \cdots \times \mathbb{Z}/(i_j)\]

EXAMPLES:

```python
sage: F.<a,b> = FreeGroup()
sage: 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` – A list/tuple/iterable with the elements of the free group.
- `**kwds` – 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
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 >
```

Division is shorthand for `quotient()`

```sage
sage: F / [a*b^2*a, b^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
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 >
```

`rank()`

Return the number of generators of self.

Alias for `ngens()`.

OUTPUT:

Integer.

EXAMPLES:

```sage
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
```

`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')
False
sage: is_FreeGroup(FreeGroup(0))
True
sage: is_FreeGroup(FreeGroup(index_set=ZZ))
True

sage.groups.free_group.wrap_FreeGroup(libgap_free_group)
Wrap a LibGAP free group.

This function changes the comparison method of libgap_free_group to comparison by Python id. If you want to put the LibGAP free group into a container (set, dict) then you should understand the implications of _set_compare_by_id(). To be safe, it is recommended that you just work with the resulting Sage \texttt{FreeGroup\_class}.

INPUT:
- libgap_free_group -- a LibGAP free group.

OUTPUT:
A Sage \texttt{FreeGroup\_class}.

EXAMPLES:
First construct a LibGAP free group:

\begin{verbatim}
sage: F = libgap.FreeGroup(['a', 'b'])
sage: type(F)
<class 'sage.libs.gap.element.GapElement'>
\end{verbatim}

Now wrap it:

\begin{verbatim}
sage: from sage.groups.free_group import wrap_FreeGroup
sage: wrap_FreeGroup(F)
Free Group on generators \{a, b\}
\end{verbatim}
Finitely presented groups are constructed as quotients of `free_group`:

```python
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 >
```

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

```python
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 >
```

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:

```python
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
```

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

```python
sage: G.gap()
```

Finitely presented groups are constructed as quotients of `free_group`:
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 ]>

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>) ]

The same holds for the group elements:

sage: G = FreeGroup(2)
sage: H = G / [G([1, 1]), G([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'>

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(M1, M1)
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):
...
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.
class `sage.groups.finitely_presented.FinitelyPresentedGroup`(free_group, relations, category=None)

Bases: `sage.groups.libgap_mixin.GroupMixinLibGAP`, `sage.structure.unique_representation.UniqueRepresentation`, `sage.groups.group.Group`, `sage.groups.libgap_wrapper.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.

**EXAMPLES:**

```python
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'>
```

**Element**

alias of `FinitelyPresentedGroupElement`

**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 \(\mathbb{Z}/(i_1) \times \cdots \times \mathbb{Z}/(i_j)\).

**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)

ALGORITHM:
Uses GAP.

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  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]
[ 0 c - c*b*c^-1*b^-1 1 - c*b*c^-1 0]
```

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()
[ -1  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]
[ 0 c - c*b*c^-1*b^-1 1 - c*b*c^-1 0]
[ 0 -t3*t4 + 1 t2 - 1 0 0]
[ -t4 + 1 0 t1 - 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
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
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?
```

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

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

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 – (default: False) boolean; if True, then attempt to reduce the presentation of the product group
- new_names – (default: True) boolean; 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([1,1,1,1])] ); 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)
```

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 >
```

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 < | >

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

AUTHORS:

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

\textbf{epimorphisms} (H)

Return the epimorphisms from \textit{self} to \textit{H}, up to automorphism of \textit{H}.

\textbf{INPUT}:

• \textit{H} – Another group

\textbf{EXAMPLES}:

sage: F = FreeGroup(3)
sage: G = F / [F([1, 2, 3, 1, 2, 3]), F([1, 1, 1])]
sage: H = AlternatingGroup(3)
sage: G.epimorphisms(H)

[[Generic morphism:
  From: Finitely presented group < x0, x1, x2 | x0^3, x0*x1*x2*x0*x1*x2, x0^3 >
  To: Alternating group of order 3!/2 as a permutation group
  Defn: x0 |--> ()
  x1 |--> (1,3,2)
  x2 |--> (1,2,3),

Generic morphism:
  From: Finitely presented group < x0, x1, x2 | x0^3, x0*x1*x2*x0*x1*x2, x0^3 >
  To: Alternating group of order 3!/2 as a permutation group
  Defn: x0 |--> (1,3,2)
  x1 |--> ()
  x2 |--> (1,2,3),

Generic morphism:
  From: Finitely presented group < x0, x1, x2 | x0^3, x0*x1*x2*x0*x1*x2, x0^3 >
  To: Alternating group of order 3!/2 as a permutation group
  Defn: x0 |--> (1,3,2)
  x1 |--> (1,2,3)
  x2 |--> ()],

(continues on next page)
ALGORITHM:
Uses libgap’s GQuotients function.

**free_group()**
Return the free group (without relations).

**OUTPUT:**
A `FreeGroup()`.

**EXAMPLES:**

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

```python
sage: G.<a,b> = FreeGroup('a, b')
sage: H = G / (a^2, b^3, a^b*b^-a^b^-b)
sage: H.order()
6
sage: F.<a,b,c> = FreeGroup()
sage: J = F / (F([1]), F([2, 2, 2]))
sage: J.order()
+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
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
```

**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
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
```

**semidirect_product**(H, hom, check=True, reduced=False)

The semidirect product of `self` with `H` via `hom`.

If there exists a homomorphism \( \phi \) from a group \( G \) to the automorphism group of a group \( H \), then we can define the semidirect product of \( G \) with \( H \) via \( \phi \) as the Cartesian product of \( G \) and \( H \) with the operation

\[
(g_1,h_1)(g_2,h_2) = (g_1\phi(g_2)(h_1)h_2).
\]

**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.
• \texttt{hom} – Homomorphism from \texttt{self} to the automorphism group of \(H\). Given as a pair, with generators of \texttt{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.

• \texttt{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.

• \texttt{reduced} – Boolean (default False). If True then the method attempts to reduce the presentation of the output group.

\textbf{OUTPUT}:

The semidirect product of \texttt{self} with \(H\) via \texttt{hom} as a finitely presented group. See \texttt{PermutationGroup_generic.semidirect_product} for a more in depth explanation of a semidirect product.

\textbf{AUTHORS}:

• Davis Shurbert (8-1-2013)

\textbf{EXAMPLES}:

Group of order 12 as two isomorphic semidirect products:

\begin{verbatim}
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)
#I Forcing finiteness test
True
\end{verbatim}

Dihedral groups can be constructed as semidirect products of cyclic groups:

\begin{verbatim}
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
\end{verbatim}

You can attempt to reduce the presentation of the output group:

\begin{verbatim}
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())])
\end{verbatim}

(continues on next page)
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

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)
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)

\textbf{simplification\_isomorphism()}

Return an isomorphism from \texttt{self} to a finitely presented group with a (hopefully) simpler presentation.

\textbf{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
Generic morphism:
  From: Finitely presented group < a, b, c | a*b*c, a*b^2, c*b*c^-2 >
  To: Finitely presented group < b | >
  Defn: a |--> b^-2
        b |--> b
        c |--> b
sage: I(a)
b^-2
sage: I(b)
b
sage: I(c)
b

\textbf{ALGORITHM:}
Uses GAP.

\textbf{simplified()}

Return an isomorphic group with a (hopefully) simpler presentation.

\textbf{OUTPUT:}
A new finitely presented group. Use \texttt{simplification\_isomorphism()} if you want to know the isomorphism.

\textbf{EXAMPLES:}

sage: G.<x,y> = FreeGroup()
sage: H = G / [x ^5, y ^4, y*x*y^3*x ^3]
A more complicated example:

```python
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, e2 >

```python
sage: H.simplified()
```

Finitely presented group < e0 | e0^2 >

### `structure_description(G, latex=False)`

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

This method 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` – a 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:**

```python
sage: G = CyclicPermutationGroup(6)
sage: G.structure_description()
'\text{C}_6'
sage: G.structure_description(latex=True)
'C_{6}'
sage: G2 = G.direct_product(G, maps=False)
sage: LatexExpr(G2.structure_description(latex=True)) \times C_{6}
```

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

```python
sage: D3 = DihedralGroup(3)
sage: D3.structure_description()
'S3'
```
We use the Sage notation for the degree of dihedral groups:

```python
sage: D4 = DihedralGroup(4)
sage: D4.structure_description()
'D4'
```

Works for finitely presented groups (trac ticket #17573):

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

And matrix groups (trac ticket #17573):

```python
sage: groups.matrix.GL(4,2).structure_description()
'A8'
```

class `sage.groups.finitely_presented.FinitelyPresentedGroupElement`(*parent, x, check=True*)

Bases: `sage.groups.free_group.FreeGroupElement`

A wrapper of GAP's Finitely Presented Group elements.

The elements are created by passing the Tietze list that determines them.

EXAMPLES:

```python
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
```

`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:

A tuple of integers.

EXAMPLES:
```python
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)
```

class `sage.groups.finitely_presented.GroupMorphismWithGensImages`

Bases: `sage.categories.morphism.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:

```python
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 >
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:

```python
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
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

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*x2, x1*x2 >
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
\begin{verbatim}
    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
    
    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}

    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^2, <identity ...> ], [ a*A, <identity ...> ],
    [ A*a, <identity ...> ], [ b^2, <identity ...> ],
    [ b*B, <identity ...> ], [ B*b, <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
    ˓→ 1, x1^3*x0^2*x1, x0*x1^2*x2 >
    with rules:
    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
    ˓→ 1, x1^3*x0^2*x1, x0*x1^2*x2 >
\end{verbatim}
with rules:
  \[x_0^{-1} \rightarrow x_0\]
  \[x_1^{-1} \rightarrow x_1\]
  \[x_0^{-2} \rightarrow 1\]
  \[x_0^{-1}x_1 \rightarrow x_2^{-1}\]
  \[x_0^{-1}x_2^{-1} \rightarrow x_1\]
  \[x_1x_0 \rightarrow x_2\]
  \[x_1^2 \rightarrow 1\]
  \[x_1x_2^{-1} \rightarrow x_1x_2\]
  \[x_1x_2 \rightarrow x_0x_2\]
  \[x_2^{-1}x_0 \rightarrow x_0x_2\]
  \[x_2^{-1}x_1 \rightarrow x_0\]
  \[x_2^{-2} \rightarrow x_2\]
  \[x_2x_0 \rightarrow x_1\]
  \[x_2x_1 \rightarrow x_0x_2\]
  \[x_2^2 \rightarrow x_2^{-1}\]

**make_confluent()**

Applies 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
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 \rightarrow 1
  b^3 \rightarrow 1
  (b*a)^2 \rightarrow 1
  a*b^3*a^-1 \rightarrow 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} \rightarrow a
  a^2 \rightarrow 1
  b^{-1}a \rightarrow a*b
```
reduce(element)
Applies 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
```

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 / \[a^3, b^2*a^2\]
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)]
```

sage.groups.finitely_presented.wrap_FpGroup(libgap_fpgroup)
Wrap a GAP finitely presented group.

This function changes the comparison method of libgap_free_group to comparison by Python id. If you want to put the LibGAP free group into a container (set, dict) then you should understand the implications of _set_compare_by_id(). To be safe, it is recommended that you just work with the resulting Sage FinitelyPresentedGroup.

INPUT:

- libgap_fpgroup – a LibGAP finitely presented group
A Sage `FinitelyPresentedGroup`.

First construct a LibGAP finitely presented group:

```python
sage: F = libgap.FreeGroup(['a', 'b'])
sage: a_cubed = F.GeneratorsOfGroup()[0] ^ 3
sage: P = F / libgap([a_cubed]); P
<fp group of size infinity on the generators [ a, b ]>
sage: type(P)
<class 'sage.libs.gap.element.GapElement'>
```

Now wrap it:

```python
sage: from sage.groups.finitely_presented import wrap_FpGroup
sage: wrap_FpGroup(P)
Finitely presented group < a, b | a^3 >
```
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`
- 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, $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$ – `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:

```python
sage: groups.presentation.Cyclic(4)
Finitely presented group < a | a^4 >
```

You can also import the desired functions:

```python
sage: from sage.groups.finitely_presented_named import CyclicPresentation
sage: CyclicPresentation(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 $n!/2$. 
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:**

```python
sage: A6 = groups.presentation.Alternating(6)
sage: A6.as_permutation_group().is_isomorphic(AlternatingGroup(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:**

```python
sage: groups.presentation.BinaryDihedral(9)
Finitely presented group < x, y, z | x^-2*y^2, x^-2*z^9, x^-1*y*z >
```

**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:**

```python
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
```

**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

\[
\langle a, x | a^{2n} = 1, x^2 = a^n, x^{-1}ax = a^{-1} \rangle
\]
Note: This group is also available as a permutation group via `groups.permutation.DiCyclic`.

EXAMPLES:

```python
sage: D = groups.presentation.DiCyclic(9); D
Finitely presented group < a, b | a^18, b^2*a^-9, b^-1*a*b*a >
```

```python
sage: D.as_permutation_group().is_isomorphic(groups.permutation.DiCyclic(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:

```python
sage: D = groups.presentation.Dihedral(7); D
Finitely presented group < a, b | a^7, b^2, (a*b)^2 >
```

```python
sage: D.as_permutation_group().is_isomorphic(DihedralGroup(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, $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$ as a finite presentation, where $n_i$ forms the invariants of the input list.

EXAMPLES:

```python
sage: groups.presentation.FGAbelian([2,2])
Finitely presented group < a, b | a^2, b^2, a^-1*b^-1*a*b >
```

```python
sage: groups.presentation.FGAbelian([2,3])
Finitely presented group < a | a^6 >
```

```python
sage: groups.presentation.FGAbelian([2,4])
Finitely presented group < a, b | a^2, b^4, a^-1*b^-1*a*b >
```

You can create free abelian groups:

```python
sage: groups.presentation.FGAbelian([0])
Finitely presented group < a | >
```

```python
sage: groups.presentation.FGAbelian([0,0])
Finitely presented group < a, b | a^-1*b^-1*a*b >
```

```python
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 >
```
And various infinite abelian groups:

```plaintext
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 >
```

Outputs are reduced to minimal generators and relations:

```plaintext
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 >
```

The trivial group is an acceptable output:

```plaintext
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 < | >
```

Input list must consist of positive integers:

```plaintext
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
```

```
```

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 \(\mathbb{Z}/p\mathbb{Z}\)

**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

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 >

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^-1*b^-1*a*b^-1 >

(continues on next page)
sage: 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:

```python
sage: S4 = groups.presentation.Symmetric(4)
sage: S4.as_permutation_group().is_isomorphic(SymmetricGroup(4))
True
```
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
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)
```

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
sage: B.<s0,s1,s2> = BraidGroup(4)
sage: s0*s1*s0
s0*s1*s0
sage: B([1,2,1])
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
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: f0 * b1
f0
sage: f0 * b1
f0
sage: f1 * b1
f1
sage: f1 * b1
f1
```

**AUTHORS:**
- Miguel Angel Marco Buzunariz
class sage.groups.braid.Braid

Bases: sage.groups.artin.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))
```

**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()
```

REFERENCES:
• [Big2003]

**TL_matrix**\(\text{(drain\_size, variab=\text{None}, sparse=True)}\)

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 \(\text{drain\_size}\), and the pairings satisfy certain rules. See \(\text{TL\_basis\_with\_drain()}\) 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:**

- \(\text{drain\_size}\) – integer between 0 and the number of strands (both inclusive)
- \(\text{variab}\) – variable (default: \(\text{None}\)); the variable in the entries of the matrices; if \(\text{None}\), then use a default variable in \(\mathbb{Z}[A, A^{-1}]\)
- \(\text{sparse}\) – boolean (default: \(\text{True}\)); whether or not the result should be given as a sparse matrix

**OUTPUT:**

The matrix of the TL representation of the braid.

The parameter \(\text{sparse}\) can be set to \(\text{False}\) 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 \(d = 0\):

```plaintext
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]
```

Test of one of the relations in \(B_8\):

```plaintext
sage: B = BraidGroup(8)
sage: d = 0
sage: B([4,5,4]).TL_matrix(d) == B([5,4,5]).TL_matrix(d)
True
```

An element of the kernel of the Burau representation, following [Big1999]:

```plaintext
sage: B = BraidGroup(6)
sage: psi1 = B([4, -5, -2, 1])
sage: psi2 = B([-4, 5, 5, 2, -1, -1])
```

(continues on next page)
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]

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 coe-
cfficient 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

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

Our last example is the Kinoshita-Terasaka knot:

sage: B = BraidGroup(4)
sage: b = B([1,1,3,2,-3,-1,2,-1,-3,-2])
sage: b.alexander_polynomial(normalized=False)
REFERENCES:
• Wikipedia article Alexander_polynomial

annular_khovanov_complex(qagrad=None, ring=Integer Ring)
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:

```python
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}
```

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 homology of that grading.
Note: This is a simple wrapper around `annular_khovanov_complex()` to compute homology from it.

EXAMPLES:

```python
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 x Z, 1: 0},
 (5, 0): {2: Z},
 (5, 2): {1: Z x Z},
 (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}
```

`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 $e_i - e_{i+1}$ 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 tuple $M, \text{Madj}, H$ is returned, where $M$ is the Burau matrix in the unitary form, $\text{Madj}$ the adjoined to $M$ and $H$ the hermitian form.

**EXAMPLES:**

```python
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]
```


\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & t^{-2} & -t^{-2} + t^{-1} & -t^{-1} + 1 \\
\end{bmatrix}
\]

\textbf{sage:} \texttt{s2.burau_matrix('x')}
\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 - x & x \\
0 & 0 & 1 & 0 \\
\end{bmatrix}
\]

\textbf{sage:} \texttt{s0.burau_matrix(reduced=True)}
\[
\begin{bmatrix}
-t & 0 & 0 \\
-t & 1 & 0 \\
-t & 0 & 1 \\
\end{bmatrix}
\]

Using the different reduced forms:

\textbf{sage:} \texttt{b.burau_matrix(reduced='simple')}
\[
\begin{bmatrix}
1 - t & -t^2 + 1 & t \\
1 - t^2 + 1 & -1 \\
1 & -t^2 + 1 & 0 \\
\end{bmatrix}
\]

\textbf{sage:} \texttt{M, Madj, H = b.burau_matrix(reduced='unitary')}
\[
\begin{bmatrix}
-t^2 + 1 & t & t^2 \\
t^2 - t & 1 - t^2 & -t^3 \\
-t^2 & -t^2 + 1 & 0 \\
\end{bmatrix}
\]

\textbf{sage:} \texttt{M * H * Madj == H}

\text{True}

REFERENCES:

- Wikipedia article Burau_representation
- [Squ1984]

\textbf{centralizer}()

Return a list of generators of the centralizer of the braid.

EXAMPLES:

\textbf{sage:} \texttt{B = BraidGroup(4)}
\textbf{sage:} \texttt{b = B([[2, 1, 3, 2]])}
\textbf{sage:} \texttt{b.centralizer()}
\texttt{[s1*s0*s2*s1, s0*s2]}

\textbf{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
• \texttt{variab} – (default: \( q \)) the variable in the resulting Laurent polynomial
• \texttt{try_inverse} – boolean (default: True); if True, attempt a faster calculation by using the inverse of the braid

\section*{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.

\section*{EXAMPLES:}

\begin{Verbatim}[commandchars=\[\]]
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
\end{Verbatim}

\section*{components_in_closure()}

Return the number of components of the trace closure of the braid.

\section*{OUTPUT:}

Positive integer.

\section*{EXAMPLES:}

\begin{Verbatim}[commandchars=\[\]]
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
\end{Verbatim}

\section*{conjugating_braid(other)}

Return a conjugating braid, if it exists.

\section*{INPUT:}

• \texttt{other} – the other braid to look for conjugating braid

\section*{EXAMPLES:}
```python
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
```

**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:**

```python
sage: B = BraidGroup(4)
sage: b = B([1, 2, -3, -2, 3, 1])
sage: db = b.deformed_burau_matrix(); db
\[
\begin{pmatrix}
ap_0^*a_p^5 & \ldots & bp_0^*ap_1^*cm_3^*bp_4 \\
\ldots \\
bp_2^*bm_3^*cp_5 & \ldots & bm_2^*am_3^*bp_4
\end{pmatrix}
\]

We check how this relates to the nondeformed Burau matrix:

```python
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()})
sage: db_simp
\[
\begin{pmatrix}
1 - 2t + t^2 & t - t^2 & t - t^2 & t^3 & t^2 - t^3
\end{pmatrix}
\]
```

(continues on next page)
sage: t = burau.parent().base_ring().gen()
sage: 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:**

```python
sage: B = BraidGroup(3)
sage: b = B([2, 2, 1, 1])
sage: c = B([2, 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
```

**is_conjugated(other)**

Check if the two braids are conjugated.

**INPUT:**

- **other** – the other braid to check for conjugacy

**EXAMPLES:**

```python
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
```

**is_periodic()**

Check whether the braid is periodic.

**EXAMPLES:**

```python
sage: B = BraidGroup(3)
sage: a = B([2, 2, -1, -1, 2, 2])
sage: b = B([2, 1, 2, 1])
sage: a.is_periodic()
False
sage: 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])  
sage: b = B([2, 1, 2, 1])
sage: a.is_pseudoanosov()  
True
sage: 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])  
sage: 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 $\mathbb{Z}[A, A^{-1}]$ 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()  
1
```

Two unlinked unknots:
The Hopf link:

```python
sage: B = BraidGroup(2)
sage: b = B([-1, -1])
sage: b.jones_polynomial()
-1/sqrt(t) - 1/t^(5/2)
```

Different representations of the trefoil and one of its mirror:

```python
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
```

K11n42 (the mirror of the “Kinoshita-Terasaka” knot) and K11n34 (the mirror of the “Conway” knot):

```python
sage: B = BraidGroup(4)
sage: b11n42 = B([1, -2, 3, -2, 3, -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())
True
```

\texttt{lcm}(other)

Return the least common multiple of the two braids.

INPUT:

- \texttt{other} – the other braid with respect with the lcm is computed

EXAMPLES:

```python
sage: B = BraidGroup(3)
sage: b = B([1, 2, -1, -2, 1])
sage: c = B([1, 2, 1])
sage: b.lcm(c)
(s0*s1)^2*s0
```
**markov_trace**(\texttt{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:**

- \texttt{variab} – variable (default: \texttt{None}); the variable in the resulting polynomial; if \texttt{None}, then use the variable $A$ in $\mathbb{Z}[A, A^{-1}]$
- \texttt{normalized} - boolean (default: \texttt{True}); if specified to be \texttt{False}, return instead a rescaled Laurent polynomial version of the Markov trace

**OUTPUT:**

If \texttt{normalized} is \texttt{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 \texttt{variab}. Otherwise it is a quotient of Laurent polynomials in \texttt{variab}.

**EXAMPLES:**

```python
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}
```

We now give the non-normalized Markov trace:

```python
sage: mt = b.markov_trace(normalized=False); mt
A^-4 + 1
```

```python
sage: mt.parent()  # Univariate Laurent Polynomial Ring in A over Integer Ring
```

**mirror_image()**

Return the image of \texttt{self} under the mirror involution (see \texttt{BraidGroup_class.mirror_involution}). The link closure of it is mirrored to the closure of \texttt{self} (see the example below of a positive amphicheiral knot).

**EXAMPLES:**

```python
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
```

**permutation()**

Return the permutation induced by the braid in its strands.

**OUTPUT:**

A permutation.

**EXAMPLES:**

```python
```
```python
sage: B.<s0, s1, s2> = BraidGroup()
sage: b = s0*s1/s2/s1
sage: b.permutation()
[4, 1, 3, 2]
sage: b.permutation().cycle_string()
'(1,4,2)'
```

```python
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:**

```python
sage: B = BraidGroup(4, 's')
sage: b = B([1, 2, 3, 1, 2, 1])
sage: b.plot()
Graphics object consisting of 30 graphics primitives
sage: b.plot(color=['red', 'blue', 'red', 'blue'])
Graphics object consisting of 30 graphics primitives
sage: B.<s, t> = BraidGroup(3)
sage: b = t^-1*s^2
sage: b.plot(orientation='left-right', color='red')
Graphics object consisting of 12 graphics primitives
```

```python
plot3d(
    color='rainbow')
```

Plots 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()  # Graphics3d Object
sage: b.plot3d(color="red")  # Graphics3d Object
sage: b.plot3d(color=["red", "blue", "red", "blue"])  # Graphics3d Object
```

```
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
```

right_normal_form()

Return the right normal form of the braid.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, 1, -2, 3, 1])
sage: 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
```

sliding_circuits()

Return the sliding circuits of the braid.

OUTPUT:

A 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],
 [s0^-1*s1^-1*s0^-2*s1^-1*s0^1*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*2*s0^2*s1^5],
 [s0^-1*s1^-1*s0^-2*s1^-1*s0*s1^2*s0^4],
 [(s0^-1*s1^-1*s0^-1)^2*s1*s0^2*s1^5*s0],
 [(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]]

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

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^-1)^2*s1^5*s0^2*s1],
 [s0^-1*s1^-1*s0^-2*s1^5*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])
(continues on next page)
```python
sage: a.thurston_type()
'pseudo-anosov'
sage: c = B([2, 1, 2, 1])
sage: c.thurston_type()
'periodic'
```

tropical_coordinates()

Return the tropical coordinates of self in the braid group $B_n$.

OUTPUT:

- a list of $2n$ tropical integers

EXAMPLES:

```python
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]
```

REFERENCES:

- [DW2007]
- [Deh2011]

ultra_summit_set()

Return a list with the orbits of the ultra summit set of self

EXAMPLES:

```python
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^-1*s1^-1*s0^-1*s1^2*s0^2*s1, (s0^-1*s1^-1*s0^-1)^2*s1*s0^2*s1^5],
 (s0^-1*s1^-1*s0^-1)^2*s1^5*s0^2*s1,
 (s0^-1*s1^-1*s0^-1)^2*s1^-1*s0^3*s1^2*s0^2*s1^2]]
```

REFERENCES:

- [DW2007]
- [Deh2011]
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:**

```python
sage: B.<a,b> = BraidGroup(); B
Braid group on 3 strands
sage: H = BraidGroup('a, b')
sage: B is H
True
sage: BraidGroup(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`

```python
sage: B = BraidGroup(3); B.generators()
(s0, s1)
sage: BraidGroup(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 trac ticket #16059):

```python
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
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):

```python
sage: F = FreeGroup(3)
sage: GF = F.cayley_graph(elements=ball(F, 4), generators=F.gens()); GF
Digraph on 40 vertices
```

**class** `sage.groups.braid.BraidGroup_class(names)`

Bases: `sage.groups.artin.FiniteTypeArtinGroup`

The braid group on \(n\) strands.

**EXAMPLES:**
```python
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
```

```python
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 non-negative 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:

A 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 $d = 3$:

```python
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]]
```

The number of basis elements hopefully corresponds to the general formula for the dimension of the representation spaces:

```python
sage: B = BraidGroup(10)
sage: d = 2
sage: B.dimension_of_TL_space(d) == len(B.TL_basis_with_drain(d))
True
```

TL_representation(drain_size, variab=None)

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 $\mathbb{Z}[A, A^{-1}]$

**OUTPUT:**
A list of matrices corresponding to the representations of each of the standard generators and their inverses.

**EXAMPLES:**

```python
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]]]
```
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
```

as_permutation_group()  
Return an isomorphic permutation group.

OUTPUT:
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
```

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
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
True
```

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:

```python
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
```

`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:

```python
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
```

`order()`

Return the number of group elements.

OUTPUT:

Infinity.

`s某些_elements()`

Return a list of some elements of the braid group.

This is used both for illustration and testing purposes.

EXAMPLES:

```python
sage: B = BraidGroup(3)
sage: B.some_elements()
[...]
```

`strands()`

Return the number of strands.

OUTPUT:

Integer.
class sage.groups.braid.MappingClassGroupAction(G, M)
Bases: sage.categories.action.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 \( \sigma_i \) 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 \texttt{mapping_class_action()} method of the braid group to construct this action.

EXAMPLES:

```python
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, \ldots, x3\}
sage: A(x0, s1)
x0
sage: A(x1, s1)
x1*x2*x1^-1
sage: A(x1^-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:

- \texttt{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:

```python
sage: B.<s0,s1,s2> = BraidGroup(4)
sage: F.<x0,x1,x2,x3> = FreeGroup(4)
sage: x0
x0
sage: x1
x1
sage: x1^-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, \ldots, x3\}
sage: A(x0, s1)
x0
sage: A(x1, s1)
x1*x2*x1^-1
sage: A(x1^-1, s1)
x1*x2^-1*x1^-1
```
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(bp_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

eps(\(N\))
Evaluate the map \(\mathcal{E}_N\) for a braid.

INPUT:
- \(N\) – an 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()[0, :]
sage: q = db.parent().base_ring().base_ring().gen()
sage: (bp_0, cp_0, ap_0,
    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)

Todo: Parallellize 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])
    bp_1, cp_1, ap_1,
Todo: Paralellize 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:

```python
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
```
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 $\mathbb{Q}$ 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 $\{s_1, \ldots, s_{m-1}\}$ are

S: $s_3 s_1 t_2 s_1^{-1} t_3 t_2 s_1^{-1} t_3^{-1} = 1$ 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).

This class implements all the groups considered by Coxeter and Assion as finitely presented groups together with the classical realizations given by the authors. It also contains the conversion maps between the two ways of realization. In addition the user 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. The methods for all this functionality are 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 as instance of `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'.
For more information type `CubicBraidGroup`?

INPUT:
- n – integer or None (default). The number of strands. This argument is passed to the corresponding argument of the classcall of `CubicBraidGroup`.
- names – string or list/tuple/iterable of strings (default:'s'). This argument is passed to the corresponding argument of the classcall of `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
```

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'.
For more information type `CubicBraidGroup`?

INPUT:
- n – integer or None (default). The number of strands. This argument is passed to the corresponding argument of the classcall of `CubicBraidGroup`.
- names – string or list/tuple/iterable of strings (default:'u'). This argument is passed to the corresponding argument of the classcall of `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);
      → U3x
Assion group on 3 strands of type U
sage: U3 == U3x
True
```

class sage.groups.cubic_braid.CubicBraidElement(parent, x, check=True)
Bases: `sage.groups.finitely_presented.FinitelyPresentedGroupElement`
This class models elements of cubic factor groups of the braid group. It is the element class of the CubicBraidGroup.
For more information see the documentation of the parent `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

braid()

Return the canonical braid preimage of self as Object of the class Braid.

OUTPUT:

The preimage of self as instance of 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

burau_matrix(\text{\texttt{root\_bur=\text{\texttt{None}}}}, \text{\texttt{domain=\text{\texttt{None}}}}, \text{\texttt{characteristic=\text{\texttt{None}}}}, \text{\texttt{var=\text{\texttt{t}}}}, \text{\texttt{reduced=\text{\texttt{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):

• \text{\texttt{root\_bur}} – six (resp. twelfth) root of unity in some field (default root of unity over $\mathbb{Q}$).

• \text{\texttt{domain}} – base_ring for the Burau matrix (default is Cyclotomic Field of order 3 and degree 2, resp. the domain of root_bur if given).

• \text{\texttt{characteristic}} - integer giving the characteristic of the domain (default is 0 or the characteristic of domain if given).

• \text{\texttt{var}} – string used for the indeterminate name in case root_bur must be constructed in a splitting field.

• \text{\texttt{reduced}} – boolean (default: False) or string; for more information see the documentation of \text{\texttt{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: 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
[ 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]
sage: BuMa.base_ring()
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: 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: 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)], reduced=unitary); BuMa
[0 I]
[I 0]
sage: BuMa.base_ring()
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: sage.groups.finitely_presented.FinitelyPresentedGroup

This class implements factor groups of the Artin braid group mapping their generators to elements of order 3 (see the module header for more information on these groups).

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 (to the constructor):

- names – see the corresponding documentation of BraidGroup_class.
- cbg_type – (optional keyword, default = CubicBraidGroup.type.Coxeter, see explanation below) of 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:

\[
S: \quad s_i s_j t_i s_j^{-1} t_j s_i^{-1} t_i^{-1} = 1 \quad \text{for } m \geq 5 \text{ in case } S(m)
\]

\[
U: \quad t_1 t_3 = 1 \quad \text{for } m \geq 5 \text{ in case } U(m)
\]

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). But anyway, the instances for CubicBraidGroup.type.Coxeter, CubicBraidGroup.type.AssionS and CubicBraidGroup.type.AssionU are different, since they have different classical realizations implemented.

The creation of instances of this class can also be done more easily by help of CubicBraidGroup(), AssionGroupS() and AssionGroupU() (similar to BraidGroup() with respect to BraidGroup_class).

EXAMPLES:

```python
sage: U3 = CubicBraidGroup(3, cbg_type=CubicBraidGroup.type.AssionU); U3
Assion group on 3 strands of type U
sage: U3.gens()
(c0, c1)
```

alternative possibilities defining U3:

```python
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)
```

alternates naming the generators:

```python
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
```

(continues on next page)
Cubic Braid group on 3 strands

```sage```
C3.gens()
(t0, t1)
```sage```
U3.is_isomorphic(C3)
#I Forcing finiteness test
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] ) 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)`

Creates an isomorphic image of `self` as a classical group according to the construction given by Coxeter resp. Assion.

INPUT (optional keyword):

• `embedded` – boolean (default = False). This boolean does effect the cases of Assion groups when they are realized as projective groups, only. More precisely: if `self` is of `cgb_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)` (`cgb_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```
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
```

(continues on next page)
Subgroup with 2 generators:
\[
\begin{bmatrix}
0 & 0 & a \\
a + 1 & a & a \\
a & a + 1 & 0 \\
\end{bmatrix}
\]
(continued from previous page)

of General Unitary Group of degree 3 over Finite Field in a of size 2^2

```python
sage: u = U3([-2,1,-2,1]); u
(u1^-1*u0)^2
```

```python
sage: uCl = U3Cl(u); uCl
(1,16)(2,9)(3,10)(4,19)(6,12)(7,20)(13,21)(14,15)
```

```python
sage: uCle = U3Clemb(u); uCle
[a + 1 a + 1 1]
[a + 1 0 a]
[ 1 a a]
```

```python
sage: U3(uCl) == u
True
```

```python
sage: U3(uCle) == u
True
```

```python
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
```

```python
sage: U3Clemb.ambient() == U4Cl
True
```

```python
sage: C4 = CubicBraidGroup(4)
sage: C4Cl = C4.as_classical_group(); C4Cl
Subgroup with 3 generators:
\[
\begin{bmatrix}
E(3)^2 & 0 & 0 \\
-E(12)^7 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]
(continued from previous page)

) of General Unitary Group of degree 3 over Universal Cyclotomic Field with respect to positive definite hermitian form
\[
\begin{bmatrix}
-E(12)^7 + E(12)^11 & 1 \\
-1 & -E(12)^7 + E(12)^11 \\
0 & -1 & -E(12)^7 + E(12)^11 \\
\end{bmatrix}
\]

```python

```python
as_matrix_group(root_bur=None, domain=None, characteristic=None, var='t', reduced=False)
```

Creates an epimorphic image of self as a matrix group by use of the burau representation.

INPUT (all arguments are optional by keyword):

- root_bur – six (resp. twelfth) root of unity in some field (default root of unity over Q).
- domain – base ring for the Burau matrix (default is Cyclotomic Field of order 3 and degree 2, resp. the domain of root_bur if given).
- characteristic - integer giving the characteristic of the domain (default is 0 or the characteristic of domain if given). If none of the keywords root_bur, domain and characteristic is 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 `bura_matrix()` of `Braid`.

**OUTPUT:**

An instance of the class `FinitelyGeneratedMatrixGroup_gap` according to the input arguments together with a group homomorphism registered as a conversion from `self` to it.

**EXAMPLES:**

```python
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]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1],

[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 0 2*t + 2 3*t + 4 0]
sage: m_back = C5(m)
sage: m_back == c
True
sage: U5 = AssionGroupU(5); U5
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
```
as_permutation_group\(\text{\texttt{use_classical=True}}\)

This method returns a permutation group isomorphic to \texttt{self} together with group isomorphism from \texttt{self} as a conversion.

INPUT (all arguments are optional by keyword):

- \texttt{use_classical} – (boolean, default True) by default the permutation group is calculated via the attached classical matrix group, since this results in a smaller degree. If set to False the permutation group will be calculated using \texttt{self} (as finitely presented group).

OUTPUT:

An instance of class \texttt{PermutationGroup\_generic} together with a group homomorphism from \texttt{self} registered as a conversion.

EXAMPLES:

```sage
sage: C3 = CubicBraidGroup(3)
sage: PC3 = C3.as_permutation_group()
sage: C3.is_isomorphic(PC3)
True
sage: PC3.degree()
8
sage: c = C3([2,1-2])
sage: C3(PC3(c)) == c
True
```

as_reflection_group()

Creates an isomorphic image of \texttt{self} as irreducible complex reflection group. This is possible only for the finite cubic braid groups of \texttt{cbg\_type CubicBraidGroup\_type.Coxeter}.

This method uses the sage implementation of reflection group via the gap3 CHEVIE package. To use this method you must have gap3 together with CHEVIE installed!

OUTPUT:

An instance of the class \texttt{IrreducibleComplexReflectionGroup} together with a group isomorphism from \texttt{self} registered as a conversion.

EXAMPLES:

```sage
sage: C3.<c1,c2> = CubicBraidGroup(3) # optional - gap3
sage: R3 = C3.as_reflection_group(); R3 # optional - gap3
Irreducible complex reflection group of rank 2 and type ST4
sage: R3.cartan_matrix() # optional - gap3
\[-2*E(3) - E(3)^2 \quad E(3)^2\]
\[-E(3)^2 -2*E(3) - E(3)^2\]
```

Conversion maps:

```sage
sage: r = R3.an_element() # optional - gap3
sage: cr = C3(r); cr # optional - gap3
```

(continues on next page)
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: C4 = CubicBraidGroup(4)          # optional - gap3
sage: R4 = C4.as_reflection_group()    # optional - gap3
sage: R4.invariant_form()             # optional - gap3
[1 0 0]
[0 1 0]
[0 0 1]
sage: _ == C4.classical_invariant_form()  # optional - gap3
False
```

`braid_group()`

Return an Instance of `BraidGroup` with identical generators, such that there exists an epimorphism to `self`.

OUTPUT:

Instance of `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

**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:

```python
sage: S15 = AssionGroupS(15)
sage: S15.order()
10977561863482259035023554842176139436811616256000
sage: C6 = CubicBraidGroup(6)
sage: 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 (optional):

- `embedded` – boolean (default = False). This boolean just effects the cases of Assion groups when they are realized as projective groups. More precisely: if `self` is of `cub_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) reps. `n = 0 mod 3` (type U) is returned. Else-wise, the centralizing element is a matrix belonging to `Sp(n,3)` reps. `GU(n,2)`.

EXAMPLES:

```python
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
```
```python
cm = U3.centralizing_element(embedded=True);
cm
[a + 1 a + 1 1]
[a + 1 0 a]
[ 1 a a]
sage: U4 = AssionGroupU(4)
sage: cm in U4Cl
True
sage: [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: C6.classical_invariant_form().is_singular()
True

cubic_braid_subgroup(nstrands=None)

Creates a cubic braid group as subgroup of self on the first nstrands strands.

INPUT:

• nstrands – integer > 0 and < self.strands() giving the number of strands for the subgroup. The default is one strand less than self has.

OUTPUT:

An instance of this class realizing the subgroup.

Note: 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

is_finite()

Method from GroupMixinLibGAP overwritten because of performance reason.

EXAMPLES:

sage: CubicBraidGroup(6).is_finite()
False
sage: AssionGroupS(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

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

class type
Bases: enum.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):
  ...
TypeError: the cbg_type must be an instance of <enum 'CubicBraidGroup.type'>
Free groups and free abelian groups implemented using an indexed set of generators.

AUTHORS:

- Travis Scrimshaw (2013-10-16): Initial version

```python
class sage.groups.indexed_free_group.IndexedFreeAbelianGroup(indices, prefix, category=None, **kwds)
```

Bases: `sage.groups.indexed_free_group.IndexedGroup, sage.groups.group.AbelianGroup`

An indexed free abelian group.

**EXAMPLES:**

```python
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'}
```

```python
class Element(F, x)
```

Bases: `sage.monoids.indexed_free_monoid.IndexedFreeAbelianMonoidElement, sage.groups.indexed_free_group.IndexedFreeGroup.Element`

**gen(x)**

The generator indexed by x of self.

**EXAMPLES:**

```python
sage: G = Groups().Commutative().free(index_set=ZZ)
sage: G.gen(0)
F[0]
sage: G.gen(2)
F[2]
```

**one()**

Return the identity element of self.

**EXAMPLES:**

```python
sage: G = Groups().Commutative().free(index_set=ZZ)
sage: G.one()
1
```
class sage.groups.indexed_free_group.IndexedFreeGroup(indices, prefix, category=None, **kwds)

Bases: sage.groups.indexed_free_group.IndexedGroup, sage.groups.group.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'}
```

class Element(F, x)

Bases: sage.monoids.indexed_free_monoid.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
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
```

`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)]
```

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]

**one()**
Return the identity element of self.

**EXAMPLES:**

```python
sage: G = Groups().free(ZZ)
sage: G.one()
1
```

---

**class** `sage.groups.indexed_free_group.IndexedGroup` *(indices, prefix, category=None, names=None, **kwds)*

**Bases:** `sage.monoids.indexed_free_monoid.IndexedMonoid`

Base class for free (abelian) groups whose generators are indexed by a set.

**gens()**
Return the group generators of self.

**EXAMPLES:**

```python
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']]
```

---

**group_generators()**
Return the group generators of self.

**EXAMPLES:**

```python
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']]
```

---

**order()**
Return the number of elements of self, which is $\infty$ unless this is the trivial group.

**EXAMPLES:**

```python
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

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

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
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

```python
class sage.groups.raag.CohomologyRAAG(R, A):
    Bases: sage.combinat.free_module.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]

```python
class Element:
    Bases: sage.algebras.clifford_algebra.CliffordAlgebraElement

    An element in the cohomology ring of a right-angled Artin group.

degree_on_basis(I)
    Return the degree on the basis element clique.
```

EXAMPLES:
```python
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}
sage: H.degree_on_basis(I)
```

(continues on next page)
sage: sorted([H.degree_on_basis(I) for I in H.basis().keys()])
[0, 1, 1, 1, 1, 2, 2]

\textbf{gen}(i)

Return the $i$-th standard generator of the algebra \texttt{self}.
This corresponds to the $i$-th vertex in the graph (under a fixed ordering of the vertices).

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

\textbf{gens}()

Return the generators of \texttt{self} (as an algebra).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C4 = graphs.CycleGraph(4)
sage: A = groups.misc.RightAngledArtin(C4)
sage: H = A.cohomology()
sage: H.gens()
(e0, e1, e2, e3)
\end{verbatim}

\textbf{ngens}()

Return the number of algebra generators of \texttt{self}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C4 = graphs.CycleGraph(4)
sage: A = groups.misc.RightAngledArtin(C4)
sage: H = A.cohomology()
sage: H.ngens()
4
\end{verbatim}

\textbf{one_basis}()

Return the basis element indexing 1 of \texttt{self}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C4 = graphs.CycleGraph(4)
sage: A = groups.misc.RightAngledArtin(C4)
sage: H = A.cohomology()
sage: H.one_basis()
()  
\end{verbatim}

\textbf{class}  \texttt{sage.groups.raag.RightAngledArtinGroup}(G, names)

\textbf{Bases:} \texttt{sage.groups.artin.ArtinGroup}

The right-angled Artin group defined by a graph $G$.  

\section{Chapter 15. Right-Angled Artin Groups}
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 $\Delta$ and $\Gamma$, 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 $\phi : A_\Gamma \to \mathbb{Z}$ by $g_v \mapsto 1$ (i.e. words of exponent sum 0).
- $A_\Gamma$ has a finite $\mathbb{K}(\pi, 1)$ space.
- $A_\Gamma$ 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 – a string or a list of generator names

**EXAMPLES:**

```sage
sage: Gamma = Graph(4)
sage: G = RightAngledArtinGroup(Gamma)
sage: a,b,c,d = G.gens()
sage: a*c*d^4*a^-3*b
v0^a*c^d^4*a^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^a^-3*b
v0^a*c^d^4*a^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()
```

(continues on next page)
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
```

REFERENCES:

- [Cha2006]
- [BB1997]
- [Dro1987]
- [CP2001]
- [VW1994]
- Wikipedia article Artin_group#Right-angled_Artin_groups

```python
class Element(parent, lst):
    Bases: sage.groups.artin.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
```

gen(i)
    Return the i-th generator of self.
```
EXAMPLES:
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.gen(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)
```

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
```

ngens()
Return the number of generators of self.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.ngens()
5
```

one()
Return the identity element 1.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.one()
1
```

one_element()
Return the identity element 1.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.one()
1
```
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

$$e^g e^h = e^{g+h} \quad \text{for all } g, h \in G.$$

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:

```python
sage: 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()`:

```python
sage: EZ = E(ZZ); EZ
Multiplicative form of Integer Ring
```

Elements of the exponentiated group can be created and manipulated as follows:

```python
sage: x = EZ(-3); x
-3
sage: x.parent()
Multiplicative form of Integer Ring
sage: EZ(-1)*EZ(6) == EZ(5)
True
sage: EZ(3)**(-1)
-3
sage: 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: E = 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]

class sage.groups.group_exp.GroupExpElement(parent, x)


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: 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

inverse()

Invert the element self.

EXAMPLES:

sage: EZ = GroupExp()(ZZ)
sage: EZ(-3).inverse()
3

class sage.groups.group_exp.GroupExp_Class(G)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Paren

The multiplicative form of a commutative additive group.
INPUT:
- \(G\): a commutative additive group

OUTPUT:
- The multiplicative form of \(G\).

EXAMPLES:

```python
sage: GroupExp()(QQ)
Multiplicative form of Rational Field
```

**Element**

alias of `GroupExpElement`

**an_element()**

Return an element of the multiplicative group.

EXAMPLES:

```python
sage: L = RootSystem(['A',2]).weight_lattice()
sage: EL = GroupExp()(L)
sage: x = EL.an_element(); x
sage: x.parent()
Multiplicative form of Weight lattice of the Root system of type ['A', 2]
```

**group_generators()**

Return generators of self.

EXAMPLES:

```python
sage: GroupExp()(ZZ).group_generators()
(1,)
```

**one()**

Return the identity element of the multiplicative group.

EXAMPLES:

```python
sage: G = GroupExp()(ZZ^2)
sage: G.one()
(0, 0)
sage: x = G.an_element(); x
(1, 0)
sage: x == x * G.one()
True
```

**product(x, y)**

Return the product of \(x\) and \(y\) in the multiplicative group.

EXAMPLES:

```python
sage: G = GroupExp()(ZZ)
sage: G.product(G(2),G(7))
9
sage: x = G(2)
```

(continues on next page)
sage: x.__mul__(G(7))
9
SEMIDIRECT PRODUCT OF GROUPS

AUTHORS:

- Mark Shimozono (2013) initial version

```python
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: sage.sets.cartesian_product.CartesianProduct

Return the semidirect product of the groups G and H using the homomorphism twist.

INPUT:

- G and H – multiplicative groups
- twist – (default: None) a function defining a homomorphism (see below)
- act_to_right – True or False (default: True)
- prefix0 – (default: None) optional string
- prefix1 – (default: None) optional string
- print_tuple – True or False (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 $G \rtimes H$ in which $G$ acts on $H$ by automorphisms. In this case there is a group homomorphism $\phi \in \text{Hom}(G, \text{Aut}(H))$ such that

$$ghg^{-1} = \phi(g)(h).$$

The homomorphism $\phi$ is specified by the input `twist`, which syntactically is the function $G \times H \to H$ defined by

$$\text{twist}(g, h) = \phi(g)(h).$$

The product on $G \ltimes H$ is defined by

$$(g_1, h_1)(g_2, h_2) = g_1h_1g_2h_2$$

$$= g_1g_2^{-1}h_1g_2h_2$$

$$= (g_1g_2, \text{twist}(g_2^{-1}, h_1)h_2)$$
If `act_to_right` is False, the group $G \rtimes H$ is specified by a homomorphism $\psi \in \text{Hom}(H, \text{Aut}(G))$ such that

$$hgh^{-1} = \psi(h)(g)$$

Then `twist` is the function $H \times G \to G$ defined by

$$\text{twist}(h, g) = \psi(h)(g).$$

so that the product in $G \rtimes H$ is defined by

$$(g_1, h_1)(g_2, h_2) = g_1h_1g_2h_2$$
$$= g_1h_1g_2h_1^{-1}h_1h_2$$
$$= (g_1\text{twist}(h_1, g_2), h_1h_2)$$

If `prefix0` (resp. `prefix1`) 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 $g \ast h$.

EXAMPLES:

```python
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\n→ on Multiplicative form of Vector space of dimension 2 over Rational Field
sage: x = H.an_element(); x
t[[(1, 0)]

sage: x^2

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)
```

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()`

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

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()

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))

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]
[0 1]
sage: one.cartesian_projection(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

(continues on next page)
sage: EL = GroupExp()(L)
sage: H = GroupSemidirectProduct(G, EL, twist = lambda g,v: EL(g*v.value),
˓→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: 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

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)]

class sage.groups.group_semidirect_product.GroupSemidirectProductElement
Bases: sage.sets.cartesian_product.CartesianProduct.Element

Element class for GroupSemidirectProduct.

inverse()
The inverse of self.

EXAMPLES:

sage: L = RootSystem(['A',2]).root_lattice()
sage: from sage.groups.group_exp import GroupExp

(continues on next page)
sage: EL = GroupExp()(L)
sage: W = L.weyl_group(prefix="s")
sage: def twist(w,v):
    ....: return EL(w.action(v.value))
sage: G = GroupSemidirectProduct(W, EL, twist, prefix1='t')
sage: g = G.an_element(); g
s1*s2 * t[2*alpha[1] + 2*alpha[2]]
sage: g.inverse()
s2*s1 * t[2*alpha[1]]

to_opposite()
Send an element to its image in the opposite semidirect product.

EXAMPLES:

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 Multiplicative form of Root lattice of the Root system of type ['A', 2])
sage: mu = L.an_element(); mu
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)
CHAPTER
EIGHTEEN

MISCELLANEOUS GROUPS

This is a collection of groups that may not fit into some of the other infinite families described elsewhere.
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 \(i\).)
- \(\alpha, \beta \in \text{Aut}(R)\)

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
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)
```

class `sage.groups.semmonomial_transformations.semmonomial_transformation_group.SemimonomialActionMat(G, M, check=True)

Bases: `sage.categories.action.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.SemimonomialActionVec(G, V, check=True)

Bases: sage.categories.action.Action

The natural left action of the semimonomial group on vectors.

The action is defined by:

\[(\alpha, \pi) \cdot (v_0, \ldots, v_{n-1}) := (\alpha(v_{\pi(1)} - 1) \cdot \phi_0^{-1}, \ldots, \alpha(v_{\pi(n-1)} - 1) \cdot \phi_{n-1}^{-1})\]

(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.SemimonomialTransformationGroup(R, len)

Bases: sage.groups.group.FiniteGroup, sage.structure.unique_representation.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 \(R^*\) of \(R\)) and the group of ring automorphisms.

The multiplication of two elements \((\phi, \pi, \alpha)(\psi, \sigma, \beta)\) with

- \(\phi, \psi \in R^*^n\)
- \(\pi, \sigma \in S_n\) (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

\[(\phi, \pi, \alpha)(\psi, \sigma, \beta) \cdot (\psi^{\pi, \alpha}, \pi \sigma, \alpha \circ \beta)\]

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), 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)
```
Element

alias of `sage.groups.semimonomial_transformations.semimonomial_transformation.SemimonomialTransformation`

base_ring()

Return the underlying ring of self.

EXAMPLES:

```sage
sage: F.<a> = GF(4)
sage: SemimonomialTransformationGroup(F, 3).base_ring() is F
True
```

degree()

Return the degree of self.

EXAMPLES:

```sage
sage: F.<a> = GF(4)
sage: SemimonomialTransformationGroup(F, 3).degree()
3
```

gens()

Return a tuple of generators of self.

EXAMPLES:

```sage
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), 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); (1,2), 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
sage: F.<a> = GF(4)
sage: SemimonomialTransformationGroup(F, 5).order() == (4-1)**5 * factorial(5)**2
True
```
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

$$(\phi, \pi, \alpha)(\psi, \sigma, \beta) = (\phi \cdot \psi^\pi \cdot \alpha, \pi \sigma, \alpha \circ \beta)$$

with $\psi^\pi \cdot \alpha = (\alpha(\psi_{(1)}^{-1}), \ldots, \alpha(\psi_{(n)}^{-1}))$ 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): trac ticket #15576 dissolve dependency on Permutations.options.mul

EXAMPLES:

```python
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)
```

class `sage.groups.semimonomial_transformations.semmonomial_transformation.SemimonomialTransformation`

Bases: `sage.structure.element.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 `SemimonomialTransformationGroup` instead.

EXAMPLES:
```python
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)
```

**get_autom()**

Returns the component corresponding to $\text{Aut}(R)$ of `self`.

EXAMPLES:

```python
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
```

**get_perm()**

Returns the component corresponding to $S_n$ of `self`.

EXAMPLES:

```python
sage: F.<a> = GF(9)
sage: SemimonomialTransformationGroup(F, 4).an_element().get_perm()
[4, 1, 2, 3]
```

**get_v()**

Returns the component corresponding to $R^\text{imes}n$ of `self`.

EXAMPLES:

```python
sage: F.<a> = GF(9)
sage: SemimonomialTransformationGroup(F, 4).an_element().get_v()
(a, 1, 1, 1)
```

**get_v_inverse()**

Returns the (elementwise) inverse of the component corresponding to $R^\text{imes}n$ of `self`.

EXAMPLES:

```python
sage: F.<a> = GF(9)
sage: SemimonomialTransformationGroup(F, 4).an_element().get_v_inverse()
(a + 2, 1, 1, 1)
```

**invert_v()**

Elementwisely invert all entries of `self` which correspond to the component $R^\text{imes}n$. 

152 Chapter 20. Elements of a semimonomial transformation group
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
```
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:

```python
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
```

`sage.groups.class_function.ClassFunction_gap(G, values)`

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:
```python
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
```

`adams_operation(k)`
Return the $k$-th Adams operation on `self`.

Let $G$ be a finite group. The $k$-th Adams operation $\Psi^k$ is given by

$$\Psi^k(\chi)(g) = \chi(g^k).$$

The Adams operations turn the representation ring of $G$ into a $\lambda$-ring.

**EXAMPLES:**

```python
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),
 (-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()`
Returns the central character of `self`.

**EXAMPLES:**

```python
sage: t = SymmetricGroup(4).trivial_character()
sage: t.central_character().values()
[1, 6, 3, 8, 6]
```

`decompose()`
Returns a list of the characters that appear in the decomposition of `chi`.

**EXAMPLES:**

```python
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))
```

`degree()`
Returns the degree of the character `self`.

**REFERENCES:**

- [Wikipedia article Adams_operation](#)
EXAMPLES:

```python
sage: S5 = SymmetricGroup(5)
sage: irr = S5.irreducible_characters()
sage: [x.degree() for x in irr]
[1, 4, 5, 6, 5, 4, 1]
```

determinant_character()

Returns the determinant character of self.

EXAMPLES:

```python
sage: t = ClassFunction(SymmetricGroup(4), [1, -1, 1, 1, -1])
sage: t.determinant_character().values()
[1, -1, 1, 1, -1]
```

domain()

Returns the domain of the self.

OUTPUT:

The underlying group of the class function.

EXAMPLES:

```python
sage: ClassFunction(SymmetricGroup(4), [1,-1,1,1,-1]).domain()
Symmetric group of order 4! as a permutation group
```

exterior_power(n)

Returns the anti-symmetrized product of self with itself \(n\) times.

INPUT:

- \(n\) – a positive integer.

OUTPUT:

The \(n\)-th anti-symmetrized power of self as a ClassFunction.

EXAMPLES:

```python
sage: chi = ClassFunction(SymmetricGroup(4), [3, 1, -1, 0, -1])
sage: p = chi.exterior_power(3)  # the highest anti-symmetric power for a 3-d
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
```

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:**

```python
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 [(4,5), (1,2), (1,2,3)] 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]
```

**irreducible_constituents()**

Returns a list of the characters that appear in the decomposition of \( \chi \).

**EXAMPLES:**

```python
sage: S5 = SymmetricGroup(5)
sage: chi = ClassFunction(S5, [22, -8, 2, 1, 1, -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, -1], [5, -1, 1, -1, -1, 1]]
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]]
```

**is_irreducible()**

Returns True if self cannot be written as the sum of two nonzero characters of self.

**EXAMPLES:**

```python
sage: S4 = SymmetricGroup(4)
sage: irr = S4.irreducible_characters()
sage: [x.is_irreducible() for x in irr]
[True, True, True, True, True]
```

**norm()**

Returns the norm of self.
EXAMPLES:

```
sage: A5 = AlternatingGroup(5)
sage: [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, -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 [(4,5), (1,2), (1,2,3)] of (Symmetric group of order 5! as a permutation group)
sage: chi.restrict(H).values()
[3, -3, -3, -1, 0, 0]
```

**scalar_product(other)**

Returns 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]]
```

**symmetric_power(n)**

Returns the symmetrized product of **self** with itself **n** times.

**INPUT:**

- **n** – a 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
```

(continues on next page)
Character of Symmetric group of order 4! as a permutation group

```python
sage: p.values()
[10, 2, -2, 1, 0]
```

tensor_product(other)

EXAMPLES:

```python
sage: S3 = SymmetricGroup(3)
sage: chi1, chi2, chi3 = S3.irreducible_characters()
sage: chi1.tensor_product(chi3).values()
[1, -1, 1]
```

values()

Return the list of values of self on the conjugacy classes.

EXAMPLES:

```python
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]]
```

class sage.groups.class_function.ClassFunction_libgap(G, values)

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:

```python
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
```

adams_operation(k)

Return the k-th Adams operation on self.

Let $G$ be a finite group. The $k$-th Adams operation $\Psi_k$ is given by

$$\Psi^k(\chi)(g) = \chi(g^k).$$

The Adams operations turn the representation ring of $G$ into a $\lambda$-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],
 [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))

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]

decompose()
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: 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]

determinant_character()
Return the determinant character of self.

EXAMPLES:

```python
sage: t = ClassFunction(SymmetricGroup(4), [1, -1, 1, 1, -1])
sage: t.determinant_character().values()
[1, -1, 1, 1, -1]
```

**domain()**

Return the domain of self.

OUTPUT:

The underlying group of the class function.

EXAMPLES:

```python
sage: ClassFunction(SymmetricGroup(4), [1,-1,1,1,-1]).domain()
Symmetric group of order 4! as a permutation group
```

**exterior_power(n)**

Return the anti-symmetrized product of self with itself n times.

INPUT:

* n – a positive integer

OUTPUT:

The n-th anti-symmetrized power of self as a `ClassFunction`.

EXAMPLES:

```python
sage: chi = ClassFunction(SymmetricGroup(4), [3, 1, -1, 0, -1])
sage: p = chi.exterior_power(3)  # the highest anti-symmetric power for a 3-d char
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
```

**gap()**

Return the underlying LibGAP element.

EXAMPLES:

```python
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_gap'>
sage: gap(chi)
ClassFunction( CharacterTable( Group( [ (1,2,3,4) ] ) ), [ 1, -1, 1, -1 ] )
sage: type(_)
<class 'sage.interfaces.gap.GapElement'>
```

**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 [(4,5), (1,2), (1,2,3)] 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]
```

$irreducible_constituents()$

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]]
```

$\text{is_irreducible()}$

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]

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]

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 [(4,5), (1,2), (1,2,3)] of (Symmetric group of order 5! as a permutation group)
sage: 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, 1, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 1]]

symmetric_power(n)
Return the symmetrized product of self with itself n times.

INPUT:

• n – a positive integer

OUTPUT:

The n-th symmetrized power of self as a ClassFunction.
EXAMPLES:

```python
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]
```

tensor_product(other)

Return the tensor product of self and other.

EXAMPLES:

```python
sage: S3 = SymmetricGroup(3)
sage: chi1, chi2, chi3 = S3.irreducible_characters()
sage: chi1.tensor_product(chi3).values()
[1, -1, 1]
```

values()

Return the list of values of self on the conjugacy classes.

EXAMPLES:

```python
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]]
```
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)
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]
[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: sage.structure.parent.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:
```
an_element()
Return a representative of self.

EXAMPLES:

```python
sage: G = SymmetricGroup(3)
sage: g = G((1,2,3))
sage: C = ConjugacyClass(G,g)
sage: C.representative()
(1,2,3)
```

is_rational()
Check if self is rational (closed for powers).

EXAMPLES:

```python
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: c = ConjugacyClass(G,g)
sage: c.is_rational()
False
```

is_real()
Check if self is real (closed for inverses).

EXAMPLES:

```python
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: c = ConjugacyClass(G,g)
sage: c.is_real()
True
```

list()
Return a list with all the elements of self.

EXAMPLES:

Groups of permutations:

```python
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
```

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)

```
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

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])]
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], [4,4], [2,1], [2,0],
        [4,1], [4,3], [4,4], [1,3], [2,4], [3,0],
        [1,4], [2,1], [3,3], [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
```

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 implemented for infinite groups! Use the iter function instead.
```

```
class sage.groups.conjugacy_classes.ConjugacyClassGAP(group, element)

Bases: sage.groups.conjugacy_classes.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:

```python
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
```

### cardinality()

Return the size of this conjugacy class.

EXAMPLES:

```python
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'>
```

### 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:

```python
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
```
23.1 Multiplicative Abelian Groups

This module lets you compute with finitely generated Abelian groups of the form

\[ G = \mathbb{Z}^r \oplus \mathbb{Z}_{k_1} \oplus \cdots \oplus \mathbb{Z}_{k_t} \]

It is customary to denote the infinite cyclic group \( \mathbb{Z} \) 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 \( t \) non-zero values. To construct this Abelian group in Sage, you can either specify all entries of \( \vec{k} \) or only the non-zero entries together with the total number of generators:

```
sage: AbelianGroup([0,0,0,2,3])
Multiplicative Abelian group isomorphic to \(
\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times C_2 \times C_3
\)

sage: AbelianGroup(5, [2,3])
Multiplicative Abelian group isomorphic to \(
\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times C_2 \times C_3
\)
```

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)
```

Note that this presentation is not unique, for example \( \mathbb{Z}_6 \cong \mathbb{Z}_2 \times \mathbb{Z}_3 \). 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 \(
C_2 \times \mathbb{Z} \times C_3 \times C_2 \times C_4
\)

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
```

(continues on next page)
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 \( \mathbb{Z}^k \to \mathbb{Z}^\ell \to A \to 1 \), for some positive integers \( k, \ell \) 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 \( \text{ord}(a_i) = p_i^{c_i} \), for some primes \( p_i \) and some positive integers \( c_i, i = 1, \ldots, \ell \). 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 \( \phi : \mathbb{Z}^\ell \to A \) is the map \( \phi(x_1, \ldots, x_\ell) = a_1^{x_1} \cdots a_\ell^{x_\ell} \), for \((x_1, \ldots, x_\ell) \in \mathbb{Z}^\ell\). The matrix of relations \( M : \mathbb{Z}^k \to \mathbb{Z}^\ell \) is the matrix whose rows generate the kernel of \( \phi \) as a \( \mathbb{Z} \)-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 \( b_1 = a_1^{f_{1,1}} \cdots a_\ell^{f_{1,\ell}}, \ldots, b_m = a_1^{f_{m,1}} \cdots a_\ell^{f_{m,\ell}} \). The kernel of the map \( \phi_B : \mathbb{Z}^m \to B \) defined by \( \phi_B(y_1, \ldots, y_m) = b_1^{y_1} \cdots b_m^{y_m} \), for \((y_1, \ldots, y_m) \in \mathbb{Z}^m\), is the kernel of the matrix
\[
F = \begin{pmatrix}
  f_{11} & f_{12} & \cdots & f_{1m} \\
  f_{21} & f_{22} & \cdots & f_{2m} \\
    \vdots & \ddots & \vdots \\
  f_{\ell,1} & f_{\ell,2} & \cdots & f_{\ell,m}
\end{pmatrix},
\]
regarded as a map \( \mathbb{Z}^m \to (\mathbb{Z}/p_1^{c_1}\mathbb{Z}) \times \cdots \times (\mathbb{Z}/p_\ell^{c_\ell}\mathbb{Z}) \). In particular, \( B \cong \mathbb{Z}^m / \ker(F) \). 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 \( S = \text{diag}(s_1, s_2, s_3, \ldots, s_r, 0, 0, \ldots, 0) \), are called determinantal divisors of \( F \). where \( r \) is the rank. The invariant factors of \( A \) are:
\[
s_1, s_2/s_1, s_3/s_2, \ldots, s_r/s_{r-1}.
\]

Sage supports multiplicative abelian groups on any prescribed finite number \( n \geq 0 \) of generators. Use the \texttt{AbelianGroup()} function to create an abelian group, and the \texttt{gen()} and \texttt{gens()} methods to obtain the corresponding generators. You can print the generators as arbitrary strings using the optional \texttt{names} argument to the \texttt{AbelianGroup()} function.

EXAMPLE 1:

We create an abelian group in zero or more variables; the syntax \texttt{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
```

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]

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

sage.groups.abeliangps.abelian_group.AbelianGroup(n, gens_orders=None, names=None)

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 – a list of non-negative integers in the form \([a_0, a_1, \ldots, a_{n-1}]\), 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 \( f_i \), as in GAP.

EXAMPLES:
class sage.groups.abelian_gps.abelian_group.AbelianGroup_class(generator_orders, names, category=None)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.groups.group.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:**

- `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)
```

(continues on next page)
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

Element
alias of sage.groups.abelian_gps.abelian_group_element.AbelianGroupElement

Subgroup
alias of AbelianGroup_subgroup
cardinality()
Return the order of this group.
EXAMPLES:

sage: G = AbelianGroup([2,3])
sage: G.order()
6
sage: G = AbelianGroup([3,2,0])
sage: 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()
Dual of Abelian Group isomorphic to Z/2Z over Cyclotomic Field of order 2 and degree 1
sage: G.dual_group().gens()
(X,)
sage: G.dual_group(names='Z').gens()
sage: G.dual_group(base_ring=QQ)
Dual of Abelian Group isomorphic to \(\mathbb{Z}/2\mathbb{Z}\) over Rational Field

### elementary_divisors()

This returns 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 \(v = \max(\text{ord}_p(a_j))\) 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:**

A tuple of integers.

**EXAMPLES:**

```python
sage: G = AbelianGroup(2, [2,3])
sage: G.elementary_divisors()
(6,)
sage: G = AbelianGroup(1, [6])
sage: G.elementary_divisors()
(6,)
sage: G = AbelianGroup(2, [2,6])
sage: G
Multiplicative Abelian group isomorphic to \(\mathbb{Z}_2 \times \mathbb{Z}_6\)
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, [6,6])
sage: G.elementary_divisors()
(6, 6)
sage: AbelianGroup([3,4,5]).elementary_divisors()
(60,)
```

### exponent()

Return the exponent of this abelian group.

**EXAMPLES:**

```python
sage: G = AbelianGroup([2,3,7]); G
Multiplicative Abelian group isomorphic to \(\mathbb{Z}_2 \times \mathbb{Z}_3 \times \mathbb{Z}_7\)
sage: G.exponent()
42
```
sage: G = AbelianGroup([2,4,6]); G
Multiplicative Abelian group isomorphic to C2 x C4 x C6

sage: 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)

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]

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:
A 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()
\begin{verbatim}
(6,)
sage: Z6.elementary_divisors()
(6,)

sage: Z2xZ3.is_isomorphic(Z6)
True
sage: Z2xZ3 is Z6
False

identity()

Return the identity element of this group.

EXAMPLES:

\begin{verbatim}
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
\end{verbatim}

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 \texttt{gens_orders()} instead:

\begin{verbatim}
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)
\end{verbatim}

Use \texttt{elementary_divisors()} if you are looking for an invariant of the group.

OUTPUT:

A tuple of integers. Zero means infinite cyclic factor.

EXAMPLES:

\begin{verbatim}
sage: J = AbelianGroup([2,3])
sage: J.invariants()
(2, 3)
\end{verbatim}
\end{verbatim}
sage: 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

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()
False
sage: T = AbelianGroup([])

23.1. Multiplicative Abelian Groups
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:

```python
sage: G1 = AbelianGroup([2,3,4,5])
sage: G2 = AbelianGroup([2,3,4,5,1])
sage: G1.is_isomorphic(G2)
True
```

is_subgroup(left, right)
Test whether left is a subgroup of right.

EXAMPLES:

```python
sage: G = AbelianGroup([2,3,4,5])
sage: G.is_subgroup(G)
True

sage: H = G.subgroup([G.1])
sage: H.is_subgroup(G)
True

sage: G.<a, b> = AbelianGroup(2)
sage: H.<c> = AbelianGroup(1)
sage: H < G
False
```

is_trivial()
Return whether the group is trivial

A group is trivial if it has precisely one element.

EXAMPLES:

```python
sage: AbelianGroup([2, 3]).is_trivial()
False
sage: AbelianGroup([1, 1]).is_trivial()
True
```

list()
Return tuple of all elements of this group.

EXAMPLES:

```python
sage: G = AbelianGroup([2,3], names = "ab")
sage: G.list()
(1, b, b^2, a, a*b, a*b^2)
```
sage: G = AbelianGroup([]); G
Trivial Abelian group
sage: G.list()
(1,)

**ngens()**
The number of free generators of the abelian group.

**EXAMPLES:**

```python
sage: F = AbelianGroup(10000)
sage: 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 $\lambda$, and using `q_subgroups_of_abelian_group()` sum the number of subgroups of type $\mu$ in an abelian $p$-group of type $\lambda$ for all $\mu$ contained in $\lambda$.

**EXAMPLES:**

```python
sage: AbelianGroup([2,3]).number_of_subgroups()
4
sage: AbelianGroup([2,0,0,3,0]).number_of_subgroups()
+Infinity
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
```

**order()**
Return the order of this group.

**EXAMPLES:**

```python
sage: G = AbelianGroup(2,[2,3])
sage: G.order()
6
```

(continues on next page)
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()
Permutation Group with generators [(3,4,5), (1,2)]
```

random_element()
Return a random element of this group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,9])
sage: G.random_element().parent() is G
True
```

subgroup(gensH, names='t')
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: 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
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\}
```

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 QQ.

EXAMPLES:

```python
sage: G = AbelianGroup([4,4])
sage: G.subgroup( [ G([1,0]), G([1,2]) ])  # Multiplicative Abelian subgroup isomorphic to C2 x C4 generated by {f0, f0*f1^2}
sage: AbelianGroup([4,4]).subgroup_reduced( [ [1,0], [1,2] ])  # 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: 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:

```python
sage: AbelianGroup([2,3]).subgroups()
[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())
81
```

torsion_subgroup()

Return the torsion subgroup of this group.

EXAMPLES:

```python
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

class sage.groups.abelian_gps.abelian_group.AbelianGroup_subgroup(ambient, gens, names='f', category=None)

Bases: sage.groups.abelian_gps.abelian_group.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: G.<a,b,c> = AbelianGroup([2,3,4])
sage: H = G.subgroup([a, b^2])
sage: 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: 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])

(continues on next page)
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        # same as A.equals(B)
False
sage: A.is_isomorphic(B)
True

gen(n)

Return the nth generator of this subgroup.

EXAMPLES:

sage: G.<a,b> = AbelianGroup(2)
sage: A = G.subgroup([a])
sage: A.gen(0)
a
gens()

Return the generators for this subgroup.

OUTPUT:

A tuple of group elements generating the subgroup.

EXAMPLES:

sage: G.<a,b> = AbelianGroup(2)
sage: A = G.subgroup([a])
sage: G.gens()
(a, b)
sage: 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)
True
sage: is_AbelianGroup(AbelianGroup(7, [3]*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 $1 < n_1 \leq n_2 \leq \ldots \leq n_k$ (indeed, these $n_i$ will all be prime powers), fix a generating set $g_i = (a_{i1}, \ldots, a_{ik})$ (with $a_{ij} \in \mathbb{Z}/n_j\mathbb{Z}$, for $1 \leq i \leq \ell$, for the group $G$, and let $d = (d_1, \ldots, d_k)$ be an element of the direct product $\mathbb{Z}/n_1\mathbb{Z} \times \ldots \times \mathbb{Z}/n_k\mathbb{Z}$. Find, if they exist, integers $c_1, \ldots, c_\ell$ such that $c_1 g_1 + \cdots + c_\ell g_\ell = d$.

In other words, solve the equation $cA = d$ for $c \in \mathbb{Z}_k$, 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 $n_1, \ldots, n_k$.

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:**

```python
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]]]
```

```python
sage: A.<a,b,c,d,e> = AbelianGroup(5,[4, 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*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^7*b*c^5*d^4*e^4, 3], [a^3*b^2*c^2*d^3*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]]]
```

**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.
23.2 Finitely generated abelian groups with GAP.

This module provides a python wrapper for abelian groups in GAP.

EXAMPLES:

```python
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: AbelianGroupGap([3,5])
Abelian group with gap, generator orders (3, 5)
```

For infinite abelian groups we use the GAP package Polycyclic:

```python
sage: AbelianGroupGap([3,0])  # optional - gap_packages
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

Bases: sage.groups.libgap_wrapper.ElementLibGAP

An element of an abelian group via libgap.

EXAMPLES:

```python
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([3,6])
sage: G.gens()
(f1, f2)
```

exponents()

Return the tuple of exponents of this element.

OUTPUT:

• a tuple of integers

EXAMPLES:

```python
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([4,7,9])
sage: gens = G.gens()
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,)
```

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)

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:
• an 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_packages
sage: g = G.gens()[0]  # optional - gap_packages
sage: g.order()  # optional - gap_packages
+Infinity

class sage.groups.abelian_gps.abelian_group_gap.AbelianGroupElement_polycyclic

Bases: sage.groups.abelian_gps.abelian_group_gap.AbelianGroupElement_gap

An element of an abelian group using the GAP package Polycyclic.

exponents()

Return the tuple of exponents of self.

OUTPUT:
• a tuple of integers

EXAMPLES:

sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([4,7,0])  # optional - gap_packages
sage: gens = G.gens()  # optional - gap_packages
sage: g.exponents()  # optional - gap_packages
(2, 4, 8)

Efficiently handles very large groups:
class sage.groups.abelian_gps.abelian_group_gap.AbelianGroupGap(generator_orders)

Bases: sage.groups.abelian_gps.abelian_group_gap.AbelianGroup_gap

Abelian groups implemented using GAP.

INPUT:

• generator_orders – a list of nonnegative integers where 0 gives a factor isomorphic to \( \mathbb{Z} \)

OUTPUT:

• an abelian group

EXAMPLES:

\[
\begin{align*}
\text{sage: } & \text{from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap} \\
\text{sage: } & \text{AbelianGroupGap([3,6])} \\
\text{sage: } & \text{Abelian group with gap, generator orders (3, 6)} \\
\text{sage: } & \text{AbelianGroupGap([3,6,5])} \\
\text{sage: } & \text{Abelian group with gap, generator orders (3, 6, 5)} \\
\text{sage: } & \text{AbelianGroupGap([3,6,0])} \quad \text{# optional - gap_packages} \\
\text{sage: } & \text{Abelian group with gap, generator orders (3, 6, 0)}
\end{align*}
\]

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: sage.groups.abelian_gps.abelian_group_gap.AbelianGroup_gap

Quotients of abelian groups by a subgroup.

Note: Do not call this directly. Instead use quotient().

EXAMPLES:

\[
\begin{align*}
\text{sage: } & \text{from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap} \\
\text{sage: } & \text{A = AbelianGroupGap([4,3])} \\
\text{sage: } & \text{N = A.subgroup([A.gen(0)^2])} \\
\text{sage: } & \text{Q1 = A.quotient(N)} \\
\text{sage: } & \text{Q1} \\
\text{sage: } & \text{Quotient abelian group with generator orders (2, 3)} \\
\text{sage: } & \text{Q2 = Q1.quotient(Q1.subgroup(Q1.gens()[1:]))} \\
\text{sage: } & \text{Q2} \\
\text{sage: } & \text{Quotient abelian group with generator orders (1, 3)}
\end{align*}
\]

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

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

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,)

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

class sage.groups.abelian_gps.abelian_group_gap.AbelianGroupSubgroup_gap(ambient, gens)
Bases: sage.groups.abelian_gps.abelian_group_gap.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 \texttt{subgroup()}.

EXAMPLES:

```python
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)
```

\texttt{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:

```python
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,)
```

\texttt{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:

```python
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
```

(continues on next page)

```python
sage: H.retract(g^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: sage.structure.unique_representation.UniqueRepresentation, sage.groups.libgap_mixin.GroupMixinLibGAP, sage.groups.libgap_wrapper.ParentLibGAP, sage.groups.group.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:**

```python
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)
```

**Element**

alias of AbelianGroupElement_gap

**all_subgroups()**

Return the list of all subgroups of this group.

**EXAMPLES:**

```python
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)]
```

**aut()**

Return the group of automorphisms of self.

**EXAMPLES:**

```python
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)
```

**automorphism_group()**

Return the group of automorphisms of self.

**EXAMPLES:**

```python
```
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)

**elementary_divisors()**

Return the elementary divisors of this group.

See `sage.groups.abelian_gps.abelian_group_gap.elementary_divisors()`.

**EXAMPLES:**

```sage
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,4,5])
sage: G.elementary_divisors()
(2, 60)
```

**exponent()**

Return the exponent of this abelian group.

**EXAMPLES:**

```sage
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
```

**gens_orders()**

Return the orders of the generators.

Use **elementary_divisors()** if you are looking for an invariant of the group.

**OUTPUT:**

• a tuple of integers

**EXAMPLES:**

```sage
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
```
identity()

Return the identity element of this group.

**EXAMPLES:**

```python
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([4,10])
sage: G.identity()
1
```

is_subgroup_of(G)

Return if self is a subgroup of G considered in the same ambient group.

**EXAMPLES:**

```python
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
go: G = AbelianGroupGap([2,3,4,5])
g: gen = G.gens()[:2]
g: S1 = G.subgroup(gen)
g: S1.is_subgroup_of(G)
True
g: S2 = G.subgroup(G.gens()[1:])
g: S2.is_subgroup_of(S1)
False
```

is_trivial()

Return True if this group is the trivial group.

**EXAMPLES:**

```python
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
go: G = AbelianGroupGap([])
g: G.is_trivial()
True
go: AbelianGroupGap([1]).is_trivial()
True
go: AbelianGroupGap([1,1,1]).is_trivial()
True
go: AbelianGroupGap([2]).is_trivial()
False
go: AbelianGroupGap([2,1]).is_trivial()
False
```

quotient(N)

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

**INPUT:**

- N – a subgroup
- check – bool (default: True) check if N is normal
EXAMPLES:

```python
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)
```

**subgroup(gens)**

Return the subgroup of this group generated by gens.

**INPUT:**
- gens – a list of elements coercible into this group

**OUTPUT:**
- a subgroup

**EXAMPLES:**

```python
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
```

### 23.3 Automorphisms of abelian groups

This implements groups of automorphisms of abelian groups.

**EXAMPLES:**

```python
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,4,5])
sage: autG = G.aut()
```

Automorphisms act on the elements of the domain:

```python
sage: g = G.an_element()
sage: f = autG.an_element()
sage: f
```

(continues on next page)
Pcgs([ f1, f2, f3 ]) -> [ f1, f1*f2*f3^2, f3^2 ]
sage: (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))
((1, 0), f1)
sage: f((1,1))
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 ])

the group order:

sage: 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

Only automorphism groups of finite abelian groups are supported:

sage: G = AbelianGroupGap([0,2]) # optional gap_packages
sage: autG = G.aut() # optional gap_packages
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: sage.groups.libgap_wrapper.ElementLibGAP

Automorphisms of abelian groups with gap.
INPUT:

- \( x \) – a libgap element
- \( \text{parent} \) – the parent \( \text{AbelianGroupAutomorphismGroup}_{\text{gap}} \)
- \( \text{check} \) – bool (default:True) checks if \( x \) is an element of the group

EXAMPLES:

```python
sage: from sage.groups.abelian_gps.abelian_group_gap import AbelianGroupGap
sage: G = AbelianGroupGap([2,3,4,5])
```

```
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:

```python
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
```

\( \text{Pcgs([ f1, f2, f3, f4 ]) -> [ f1*f4, f2^2, f1*f3, f4 ]} \)

```
sage: f.matrix()
```

\[
\begin{bmatrix}
1 & 0 & 2 \\
0 & 2 & 0 \\
1 & 0 & 1
\end{bmatrix}
\]

Compare with the exponents of the images:

```python
sage: f(G.gens()[0]).exponents()
```

(1, 0, 2)

```python
sage: f(G.gens()[1]).exponents()
```

(0, 2, 0)

```python
sage: f(G.gens()[2]).exponents()
```

(1, 0, 1)

```python
class sage.groups.abelian_gps.abelian_aut.AbelianGroupAutomorphismGroup(AbelianGroupGap)
Bases: sage.groups.abelian_gps.abelian_aut.AbelianGroupAutomorphismGroup_{gap}
```

The full automorphism group of a finite abelian group.

INPUT:

- AbelianGroupGap – an instance of \( \text{AbelianGroup\_gap} \)

EXAMPLES:

```python
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])
```
Element alias of \texttt{AbelianGroupAutomorphism}

class \texttt{sage.groups.abelian_gps.abelian_aut.AbelianGroupAutomorphismGroup\_gap}(\texttt{domain}, \texttt{gap\_parent}, \texttt{category}, \texttt{ambient=None})

Bases: \texttt{sage.structure.unique\_representation.CachedRepresentation}, \texttt{sage.groups.libgap\_mixin.GroupMixInLibGAP}, \texttt{sage.groups.group.Group}, \texttt{sage.groups.libgap\_wrapper.ParentLibGAP}

Base class for groups of automorphisms of abelian groups.

Do not construct this directly.

INPUT:

• \texttt{domain} – \texttt{AbelianGroup\_gap}

• \texttt{libgap\_parent} – the libgap element that is the parent in GAP

• \texttt{category} – a category

• \texttt{ambient} – an instance of a derived class of \texttt{ParentLibGAP} or \texttt{None} (default); the ambient group if \texttt{libgap\_parent} has been defined as a subgroup

EXAMPLES:

```python
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>
```

Element alias of \texttt{AbelianGroupAutomorphism}

covering\_matrix\_ring()

Return the covering matrix ring of this group.

This is the ring of \( n \times n \) matrices over \( \mathbb{Z} \) where \( n \) is the number of (independent) generators.

EXAMPLES:

```python
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
```

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)
```

**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
```

class sage.groups.abelian_gps.abelian_aut.AbelianGroupAutomorphismGroup_subgroup(ambient, generators)

Bases: sage.groups.abelian_gps.abelian_aut.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` – a 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)
```

**Element**

alias of AbelianGroupAutomorphism
23.4 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 $\mathbb{Z}_6$ 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
sage: (g*g).value()
1
```

The elements come with a coercion embedding into the `values_group()`, so you can use the group elements instead of the values:

```
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
```
Construct an Abelian group with values associated to the generators.

**INPUT:**

- `values` – a 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` – a list of non-negative integers in the form \([a_0, a_1, \ldots, a_{n-1}]\), 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:**

```python
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)
```

The group elements come with a coercion embedding into the `values_group()`, so you can use them like their `value()`

```python
sage: G.values_group()
```

```output
Integer Ring
```

```python
sage: G.values_embedding()
```

```output
Generic morphism:
  From: Multiplicative Abelian group isomorphic to C6
  To:   Integer Ring
```

```python
sage: g.value()
```

```output
-1
```

```python
sage: 0 + g
```

```output
-1
```

```python
sage: 1 + 2*g
```

```output
-1
```

### class `sage.groups.abelian_gps.values.AbelianGroupWithValuesElement` (parent, exponents, value=None)

**Bases:** `sage.groups.abelian_gps.abelian_group_element.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:

```python
sage: F = AbelianGroupWithValues([1,-1], [2,4])
sage: a,b = F.gens()
sage: TestSuite(a*b).run()
```

inverse()

Return the inverse element.

EXAMPLES:

```python
sage: G.<a,b> = AbelianGroupWithValues([2,-1], [0,4])
sage: a.inverse()
a^-1
sage: a.inverse().value()
1/2
sage: a.__invert__().value()
1/2
sage: (~a).value()
1/2
sage: (a*b).value()
-2
sage: (a*b).inverse().value()
-1/2
```

value()

Return the value of the group element.

OUTPUT:

The value according to the values for generators, see gens_values().

EXAMPLES:

```python
sage: G = AbelianGroupWithValues([5], 1)
sage: G.0.value()
5
```

class sage.groups.abelian_gps.values.AbelianGroupWithValuesEmbedding(domain, codomain)

Bases: sage.categories.morphism.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:

```python
sage: Z4.<g> = AbelianGroupWithValues([I], [4])
sage: embedding = Z4.values_embedding(); embedding
Generic morphism:
```
From: Multiplicative Abelian group isomorphic to $\mathbb{C}_4$
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

class sage.groups.abelian_gps.values.AbelianGroupWithValues_class(generator_orders, names, values, values_group)
Bases: sage.groups.abelian_gps.abelian_group.AbelianGroup_class
The class of an Abelian group with values associated to the generator.

INPUT:
- 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()

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
sage: G = AbelianGroupWithValues([-1,0,1], [2,1,3])
sage: G.gens()
(f0, 1, f2)
**gens_values()**

Return the values associated to the generators.

OUTPUT:

A tuple.

EXAMPLES:

```
sage: G = AbelianGroupWithValues([-1,0,1], [2,1,3])
sage: G.gens()  
(\text{f0}, 1, \text{f2})
sage: G.gens_values()  
(-1, 0, 1)
```

**values_embedding()**

Return the embedding of \texttt{self} in \texttt{values_group()}.

OUTPUT:

A morphism.

EXAMPLES:

```
sage: Z4 = AbelianGroupWithValues([I], [4])
sage: Z4.values_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
```

**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 \texttt{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

def G: Z4 = AbelianGroupWithValues([I], [4])
sage: Z4.values_group()  
Number Field in I with defining polynomial x^2 + 1 with I = 1*I
```
23.5 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 $\mathbb{C}^\times$). Let $g_j, j = 1, \ldots, n$, denote generators of $G$ - say $g_j$ is of order $m_j > 1$. There are generators $X_j, j = 1, \ldots, n$, 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 $\text{AbelianGroup()}$ function to create an abelian group, the $\text{dual_group()}$ method to create its dual, and then the $\text{gen()}$ and $\text{gens()}$ methods to obtain the corresponding generators. You can print the generators as arbitrary strings using the optional names argument to the $\text{dual_group()}$ method.

**EXAMPLES:**

```python
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: Fd = F.dual_group(names='ABCDE', base_ring=CC)
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
```

**AUTHORS:**
- David Joyner (2006-08) (based on abelian_groups)
- David Joyner (2006-10) modifications suggested by William Stein

```
class sage.groups.abelian_gps.dual_abelian_group.DualAbelianGroup_class(G, names, base_ring)
    Bases: sage.structure.unique_representation.UniqueRepresentation, sage.groups.group.AbelianGroup

    Dual of abelian group.

    EXAMPLES:
```
Dual of Abelian Group isomorphic to $\mathbb{Z}/15\mathbb{Z} \times \mathbb{Z}/7\mathbb{Z} \times \mathbb{Z}/8\mathbb{Z} \times \mathbb{Z}/9\mathbb{Z}$ over Complex Field with 53 bits of precision

**Element**
alias of `sage.groups.abelian_gps.dual_abelian_group_element.DualAbelianGroupElement`

**base_ring()**
Return the scalars over which the group is dualized.

**EXAMPLES:**

```python
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
```

**gen(i=0)**
The $i$-th generator of the abelian group.

**EXAMPLES:**

```python
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)
```

**gens()**
Return the generators for the group.

**OUTPUT:**
A tuple of group elements generating the group.

**EXAMPLES:**

```python
sage: F = AbelianGroup([7, 11]).dual_group()
sage: F.gens()
(X0, X1)
```

**gens_orders()**
The orders of the generators of the dual group.

**OUTPUT:**
A tuple of integers.

**EXAMPLES:**

```python
sage: F = AbelianGroup([5]*1000)
sage: Fd = F.dual_group()
sage: invs = Fd.gens_orders(); len(invs)
1000
```
### group()
Return the group that `self` is the dual of.

**EXAMPLES:**

```python
sage: F = AbelianGroup([3,[5,64,729], names=list("abc"))
sage: Fd = F.dual_group(base_ring=CC)
sage: Fd.group() is F
True
```

### invariants()
The invariants of the dual group.
You should use `gens_orders()` instead.

**EXAMPLES:**

```python
sage: F = AbelianGroup([5]*1000)
sage: Fd = F.dual_group()
$$sage: invs = Fd.gens_orders(); len(invs)
1000$$
```

### is_commutative()
Return True since this group is commutative.

**EXAMPLES:**

```python
sage: G = AbelianGroup([2,3,9])
sage: Gd = G.dual_group()
sage: Gd.is_commutative()
True
sage: Gd.is_abelian()
True
```

### list()
Return tuple of all elements of this group.

**EXAMPLES:**

```python
sage: G = AbelianGroup([2,3], names="ab")
sage: Gd = G.dual_group(names="AB")
$$sage: Gd.list()
```

### ngens()
The number of generators of the dual group.

**EXAMPLES:**

```python
sage: F = AbelianGroup([7]*100)
sage: Fd = F.dual_group()
sage: Fd.ngens()
100
```

### order()
Return the order of this group.

**EXAMPLES:**

```python
```
random_element()

Return a random element of this dual group.

EXAMPLES:

```python
sage: G = AbelianGroup([2,3,9])
sage: Gd = G.dual_group()
sage: Gd.order()
54
```

sage.groups.abelian_gps.dual_abelian_group.is_DualAbelianGroup(x)

Return True if \(x\) is the dual group of an abelian group.

EXAMPLES:

```python
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)
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)
```

## 23.6 Base class for abelian group elements

This is the base class for both \texttt{abelian\_group\_element} and \texttt{dual\_abelian\_group\_element}.

As always, elements are immutable once constructed.

```python
class sage.groups.abelian_gps.element_base.AbelianGroupElementBase(parent, exponents)
Bases: sage.structure.element.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 \(G\). \(\text{gen}(i).\text{order}()\) (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")
sage: A,B,C = Fd.gens()
sage: A*B^-1 in Fd
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)
```

**inverse()**

Returns the inverse element.

**EXAMPLES:**

```
sage: G.<a,b> = AbelianGroup([0,5])
sage: a.inverse()
a^-1
sage: a.__invert__()
a^-1
sage: ~a
a^-1
sage: (a*b).exponents()
(1, 1)
sage: (a*b).inverse().exponents()
(-1, 4)
```
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
```

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: 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()
(1, 1, 2, 6, 1)
sage: X = A*B*C^2*D^2*E^-6
sage: X.exponents()
(1, 1, 2, 2, 2)
```

multiplicative_order()
Return the order of this element.

OUTPUT:
An integer or infinity.

EXAMPLES:

```
sage: F = AbelianGroup(3,[7,8,9])
sage: Fd = F.dual_group()
sage: A,B,C = Fd.gens()
sage: (B*C).order()
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
```

(continues on next page)
order()

Return the order of this element.

OUTPUT:

An integer or infinity.

EXAMPLES:

```python
sage: F = AbelianGroup(3,[7,8,9])
sage: Fd = F.dual_group()
sage: A,B,C = Fd.gens()
sage: (B*C).order()
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
```

23.7 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:

```python
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
(continues on next page)
```
class sage.groups.abelian_gps.abelian_group_element.AbelianGroupElement(parent, exponents)

Bases: sage.groups.abelian_gps.element_base.AbelianGroupElementBase

Elements of an `AbelianGroup`

INPUT:

- `x` – list/tuple/iterable of integers (the element vector)
- `parent` – the parent `AbelianGroup`

EXAMPLES:

```python
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^2 * b^3
```

`as_permutation()`

Return the element of the permutation group G (isomorphic to the abelian group A) associated to a in A.

EXAMPLES:

```python
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
Permutation Group with generators [(6,7,8,9), (3,4,5), (1,2)]
sage: a.as_permutation()
(1,2)
sage: ap = a.as_permutation(); ap
(1,2)
sage: ap in Gp
True
```

`word_problem(words)`

TODO - this needs a rewrite - see stuff in the matrix_grp directory.

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.

**EXAMPLES:**

```python
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
```

`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:

```python
sage: from sage.groups.abelian_gps.abelian_group_element import is_AbelianGroupElement
sage: is_AbelianGroupElement(3)
False
```

23.8 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:

```python
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")
sage: 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:

```python
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
```

(continues on next page)
AUTHORS:
- David Joyner (2006-07); based on abelian_group_element.py.
- David Joyner (2006-10); modifications suggested by William Stein.

class sage.groups.abelian_gps.dual_abelian_group_element.DualAbelianGroupElement(parent, exponents)

Bases: sage.groups.abelian_gps.element_base.AbelianGroupElementBase

Base class for abelian group elements

word_problem(words, display=True)

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, 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],display=False)=[[b^2*c^2*d^3*e^5, 245]]
The command e.word_problem([u,v,w,x,y],display=True) returns the same list but also prints $e = (b^2*c^2*d^3*e^5)^{245}$.

sage.groups.abelian_gps.dual_abelian_group_element.add_strings(x, z=0)

This was in sage.misc.misc but commented out. Needed to add lists of strings in the word_problem method below.

Return the sum of the elements of x. If x is empty, return z.
INPUT:
- x – iterable
- z – the θ that will be returned if x is empty.

OUTPUT:
The sum of the elements of x.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.dual_abelian_group_element import add_strings
sage: add_strings([], z='empty')
'empty'
sage: add_strings(['a', 'b', 'c'])
'abc'
```

```
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_element import is_DualAbelianGroupElement
sage: F = AbelianGroup(5,[5,5,7,8,9],names = list("abcde")).dual_group()
sage: is_DualAbelianGroupElement(F)
False
sage: is_DualAbelianGroupElement(F.an_element())
True
```

### 23.9 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: sage.categories.morphism.Morphism
A set-theoretic map between AbelianGroups.
```

```
class sage.groups.abelian_gps.abelian_group_morphism.AbelianGroupMorphism(G, H, genss, imgss)
Bases: sage.categories.morphism.Morphism
```

23.9. Homomorphisms of abelian groups 215
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:

```python
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])
```

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:

```python
sage: G = AbelianGroup(2,[2,3],names="xy")
sage: x,y = G.gens()
sage: subG = G.subgroup([x])
sage: H = AbelianGroup(3,[2,3,4],names="abc")
sage: a,b,c = H.gens()
sage: phi = AbelianGroupMorphism(H,G,[x,y],[a,b])
sage: phi.image(subG)
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:

```python
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])
sage: phi.kernel()
Group([ ])
```
Additive abelian groups are just modules over \( \mathbb{Z} \). 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.

```python
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): 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 \( \bigoplus_i \mathbb{Z}/n_i \mathbb{Z} \), where \( n_i \) are the invariants.

**EXAMPLES:**

```python
sage: AdditiveAbelianGroup([0, 2, 4])
Additive abelian group isomorphic to \( \mathbb{Z} + \mathbb{Z}/2 + \mathbb{Z}/4 \)
```

An example of the `remember_generators` switch:

```python
sage: G = AdditiveAbelianGroup([0, 2, 3]); G
Additive abelian group isomorphic to \( \mathbb{Z} + \mathbb{Z}/2 + \mathbb{Z}/3 \)
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 \( \mathbb{Z}/6 + \mathbb{Z} \)
sage: 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

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 = 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
class sage.groups.additive_abelian.additive_abelian_group.AdditiveAbelianGroupElement(parent, x, check=True)

Bases: sage.modules.fg_pid.fgp_element.FGP_Element

An element of an AdditiveAbelianGroup_class.

class sage.groups.additive_abelian.additive_abelian_group.AdditiveAbelianGroup_class(cover, relations)

Bases: sage.modules.fg_pid.fgp_module.FGP_Module_class, sage.groups.old.AbelianGroup

An additive abelian group, implemented using the \( \mathbb{Z} \)-module machinery.

INPUT:

• cover – the covering group as \( \mathbb{Z} \)-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:

```python
sage: AdditiveAbelianGroup([2,4]).exponent()
4
sage: AdditiveAbelianGroup([0, 2,4]).exponent()
0
sage: AdditiveAbelianGroup([]).exponent()
1
```

is_cyclic()
Returns True if the group is cyclic.

EXAMPLES:

With no common factors between the orders of the generators, the group will be cyclic.

```python
sage: G=AdditiveAbelianGroup([6, 7, 55])
sage: G.is_cyclic()
True
```

Repeating primes in the orders will create a non-cyclic group.

```python
sage: G=AdditiveAbelianGroup([6, 15, 21, 33])
sage: G.is_cyclic()
False
```

A trivial group is trivially cyclic.

```python
sage: T=AdditiveAbelianGroup([1])
sage: T.is_cyclic()
True
```
is_multiplicative()

Return False since this is an additive group.

EXAMPLES:

```
sage: AdditiveAbelianGroup([0]).is_multiplicative()
False
```

order()

Return the order of this group (an integer or infinity)

EXAMPLES:

```
sage: AdditiveAbelianGroup([2,4]).order()
8
sage: AdditiveAbelianGroup([0, 2,4]).order()
+Infinity
sage: 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'
```

class sage.groups.additive_abelian.additive_abelian_group.AdditiveAbelianGroup_fixed_gens(cover, rels, gens)

Bases: sage.groups.additive_abelian.additive_abelian_group.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),)
```

identity()

Return the identity (zero) element of this group.

EXAMPLES:

```
sage: G = AdditiveAbelianGroup([2, 3])
sage: G.identity()
(0, 0)
```

permutation_group()

Return the permutation group attached to this group.
EXAMPLES:

```
sage: G = AdditiveAbelianGroup([2, 3])
sage: G.permutation_group()
Permutation Group with generators [(3,4,5), (1,2)]
```

`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 \( \mathbb{Z} \) 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])
```

### 23.11 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 \( \mathbb{Q} \):

```
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])
sage: M
Additive abelian group isomorphic to \( \mathbb{Z}/3 + \mathbb{Z}/2 + \mathbb{Z}/2 \) 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, see https://trac.sagemath.org/sage_trac/ticket/11599#comment:7
True
```

We check that ridiculous operations are being avoided:

```
23.11. Wrapper class for abelian groups 221
```
Todo:

- Implement proper black-box discrete logarithm (using baby-step giant-step). The discrete_exp function can also potentially be speeded up substantially via caching.
- 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.

class sage.groups.additive_abelian.additive_abelian_wrapper.AdditiveAbelianGroupWrapper

Bases: sage.groups.additive_abelian.additive_abelian_group.AdditiveAbelianGroup_fixed_gens

The parent of AdditiveAbelianGroupWrapperElement

Element

alias of AdditiveAbelianGroupWrapperElement

generator_orders()

The orders of the generators with which this group was initialsed. (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,)
```

universe()

The ambient group in which this abelian group lives.

EXAMPLES:

```
sage: G = AdditiveAbelianGroupWrapper(QQbar, [sqrt(QQbar(2)), sqrt(QQbar(3))], ...
˓→[0, 0])
sage: G.universe()
Algebraic Field
class sage.groups.additive_abelian.additive_abelian_wrapper.AdditiveAbelianGroupWrapperElement

Bases: sage.groups.additive_abelian.additive_abelian_group.AdditiveAbelianGroupElement

An element of an AdditiveAbelianGroupWrapper.

element()

Return the underlying object that this element wraps.

EXAMPLES:

```python
sage: T = EllipticCurve('65a').torsion_subgroup().gen(0)
sage: T; type(T)
(0 : 0 : 1)
<class 'sage.schemes.elliptic_curves.ell_torsion.EllipticCurveTorsionSubgroup_with_category.element_class'>
sage: T.element(); type(T.element())
(0 : 0 : 1)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field'>
```

class sage.groups.additive_abelian.additive_abelian_wrapper.UnwrappingMorphism(domain)

Bases: sage.categories.morphism.Morphism

The embedding into the ambient group. Used by the coercion framework.

### 23.12 Groups of elements representing (complex) arguments.

This includes

- `RootsOfUnityGroup` (containing all roots of unity)
- `UnitCircleGroup` (representing elements on the unit circle by $e^{2\pi \cdot exponent}$)
- `ArgumentByElementGroup` (whose elements are defined via formal arguments by $e^{I \cdot arg(element)}$)

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)
23.12.1 Classes and Methods

class sage.groups.misc_gps.argument_groups.AbstractArgument(parent, element, normalize=True)
   Bases: sage.structure.element.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 – a boolean (default: True)

class sage.groups.misc_gps.argument_groups.AbstractArgumentGroup(base, category)
   Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.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: sage.groups.misc_gps.argument_groups.AbstractArgument

   An element of ArgumentByElementGroup.

   INPUT:
   - parent – a SageMath parent
   - element – a nonzero element of the parent's base
   - normalize – a boolean (default: True)

class sage.groups.misc_gps.argument_groups.ArgumentByElementGroup(base, category)
   Bases: sage.groups.misc_gps.argument_groups.AbstractArgumentGroup

   A group of (complex) arguments. The arguments are represented by 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)
e^(I*arg(1.00000000000000 + 2.00000000000000*I))
Element
alias of ArgumentByElement

sage.groups.misc_gps.argument_groups.ArgumentGroup =
<__main__.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: sage.structure.factory.UniqueFactory
A factory for creating argument groups.

INPUT:

• data – an object
  The factory will analyze data and interpret it as specification or domain.
• specification – a 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')
Group of Roots of Unity
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)
Sign Group
sage: ArgumentGroup('Arg_RR')
Sign Group
```
sage: ArgumentGroup(RIF)
Sign Group
sage: ArgumentGroup(RBF)
Sign Group
sage: ArgumentGroup(CC)
Unit Circle Group with Exponents in
Real Field with 53 bits of precision modulo ZZ
sage: ArgumentGroup('Arg_CC')
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))
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: sage.groups.misc_gps.argument_groups.UnitCirclePoint

A root of unity (i.e. an element of RootsOfUnityGroup) which is $e^{2\pi \cdot \text{exponent}}$ 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

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

(continues on next page)
class sage.groups.misc_gps.argument_groups.RootsOfUnityGroup(category)
Bases: sage.groups.misc_gps.argument_groups.UnitCircleGroup
The group of all roots of unity.
INPUT:

• category – a category
This is a specialized UnitCircleGroup with base \( \mathbb{Q} \).
EXAMPLES:

```python
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
```

Element
alias of RootOfUnity
class sage.groups.misc_gps.argument_groups.Sign(parent, element, normalize=True)
Bases: sage.groups.misc_gps.argument_groups.AbstractArgument
An element of SignGroup.
INPUT:

• parent – a SageMath parent
• element – a nonzero element of the parent’s base
• normalize – a boolean (default: True)
is_minus_one()
Return whether this sign is \(-1\).
EXAMPLES:

```python
sage: from sage.groups.misc_gps.argument_groups import SignGroup
sage: S = SignGroup()
sage: S(-1).is_minus_one()
True
```
is_one()
Return whether this sign is 1.
EXAMPLES:

```python
sage: from sage.groups.misc_gps.argument_groups import SignGroup
sage: S = SignGroup()
sage: S(-1).is_one()
False
```
class sage.groups.misc_gps.argument_groups.SignGroup(category)
Bases: sage.groups.misc_gps.argument_groups.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
```

Element
alias of Sign
class sage.groups.misc_gps.argument_groups.UnitCircleGroup(base, category)
Bases: sage.groups.misc_gps.argument_groups.AbstractArgumentGroup

A group of points on the unit circle. These points are represented by $e^{2\pi \cdot \text{exponent}}$.

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)
```

Element
alias of UnitCirclePoint
class sage.groups.misc_gps.argument_groups.UnitCirclePoint(parent, element, normalize=True)
Bases: sage.groups.misc_gps.argument_groups.AbstractArgument

An element of UnitCircleGroup which is $e^{2\pi \cdot \text{exponent}}$.

INPUT:

• parent – a SageMath parent
• exponent – a number (of a subset of the reals)
• normalize – a boolean (default: True)

exponent
The exponent of this point on the unit circle.

EXAMPLES:

```python
sage: from sage.groups.misc_gps.argument_groups import UnitCircleGroup
sage: C = UnitCircleGroup(RR)
sage: C(exponent=4/3).exponent
0.33333333333333333333
```

is_minus_one()
Return whether this point on the unit circle is $-1$.

EXAMPLES:

```python
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
```

is_one()
Return whether this point on the unit circle is 1.

EXAMPLES:

```python
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
```

### 23.13 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 $n^{\Re z} + i \cdot n^{\Im z}$. (Note that the first summand in the exponent determines the growth, the second does not influence the growth.)

**AUTHORS:**

- Daniel Krenn (2018)
23.13.1 Classes and Methods

class sage.groups.misc_gps.imaginary_groups.ImaginaryElement(parent, imag)
Bases: sage.structure.element.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()  # indirect doctest
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()  # indirect doctest
0

class sage.groups.misc_gps.imaginary_groups.ImaginaryGroup(base, category)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.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)  # indirect doctest
0
sage: J(img=100)
100*I
sage: J(3*I)
3*I
sage: J(1+2*I)
(continues on next page)
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*
24.1 Catalog of permutation groups

Type `groups.permutation.<tab>` to access examples of groups implemented as permutation groups.

24.2 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`.

```python
sage.groups.perm_gps.constructor.PermutationGroupElement(g, parent=None, check=True)
```

Builds 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` – (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:

```python
sage: p = PermutationGroupElement([1,4,2,3])
sage: p
(2,4,3)
sage: p.parent()
Symmetric group of order 4! as a permutation group
```

Initialization as a list of cycles:
Initialization as a string representing a cycle decomposition:

```python
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
```

By default the constructor assumes that the domain is \{1, \ldots, n\} but it can be set to anything via its second parent argument:

```python
sage: S = SymmetricGroup(['a', 'b', 'c', 'd', 'e'])
sage: p = PermutationGroupElement(['e', 'c', 'b', 'a', 'd'], S)
('a','e','d')('b','c')
sage: p
(a,b,c)
sage: p.parent()
Symmetric group of order 5! as a permutation group
```

But in this situation, you might want to use the more direct:

```python
sage: S(('a','b','e'))
(a,b,e)(c,d)
```

**sage.groups.perm_gps.constructor.standardize_generator**

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, Permutation
- `convert_dict` – (optional) a dictionary used to convert the points to a number compatible with GAP
- `as_cycles` – (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_())
[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,)]

sage: d = {'a': 1, 'b': 2}

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)]
```

```
sage.groups.perm_gps.constructor.string_to_tuples(g)
EXAMPLES:
```
### 24.3 Permutation groups

A permutation group is a finite group $G$ whose elements are permutations of a given finite set $X$ (i.e., bijections $X \rightarrow X$) 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:

\[(a,\ldots,b)(c,\ldots,d)\ldots(e,\ldots,f) \iff [(a,\ldots,b), (c,\ldots,d),\ldots, (e,\ldots,f)] \]

\[() = \text{identity} \iff [] \]

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.


#### 24.3.1 Index of methods

Here are the method of a `PermutationGroup()`

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>as_finitely_presented_group()</code></td>
<td>Return a finitely presented group isomorphic to self.</td>
</tr>
<tr>
<td><code>blocks_all()</code></td>
<td>Returns the list of block systems of imprimitivity.</td>
</tr>
<tr>
<td><code>cardinality()</code></td>
<td>Return the number of elements of this group. See also: G.degree()</td>
</tr>
<tr>
<td><code>center()</code></td>
<td>Return the subgroup of elements that commute with every element of this group.</td>
</tr>
<tr>
<td><code>centralizer()</code></td>
<td>Returns the centralizer of $g$ in self.</td>
</tr>
<tr>
<td><code>character()</code></td>
<td>Returns a group character from values, where values is a list of the values of the character evaluated on the conjugacy classes.</td>
</tr>
<tr>
<td><code>character_table()</code></td>
<td>Returns the matrix of values of the irreducible characters of a permutation group $G$ at the conjugacy classes of $G$.</td>
</tr>
<tr>
<td><code>cohomology()</code></td>
<td>Computes the group cohomology $H^n(G, F)$, where $F$ = $\mathbb{Z}$ if $p = 0$ and $F$ = $\mathbb{Z}$/$p\mathbb{Z}$ if $p &gt; 0$ is a prime.</td>
</tr>
<tr>
<td><code>cohomology_part()</code></td>
<td>Compute the $p$-part of the group cohomology $H^n(G, F)$, where $F$ = $\mathbb{Z}$ if $p = 0$ and $F$ = $\mathbb{Z}$/$p\mathbb{Z}$ if $p &gt; 0$ is a prime.</td>
</tr>
<tr>
<td><code>commutator()</code></td>
<td>Returns the commutator subgroup of a group, or of a pair of groups.</td>
</tr>
<tr>
<td><code>composition_series()</code></td>
<td>Return the composition series of this group as a list of permutation groups.</td>
</tr>
<tr>
<td><code>conjugacy_class()</code></td>
<td>Return the conjugacy class of $g$ inside the group self.</td>
</tr>
</tbody>
</table>

continues on next page
### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>conjugacy_classes()</code></td>
<td>Return a list with all the conjugacy classes of <code>self</code>.</td>
</tr>
<tr>
<td><code>conjugacy_classes_representatives()</code></td>
<td>Returns a complete list of representatives of conjugacy classes in a permutation group <code>G</code>.</td>
</tr>
<tr>
<td><code>conjugacy_classes_subgroups()</code></td>
<td>Returns a complete list of representatives of conjugacy classes of subgroups in a permutation group <code>G</code>.</td>
</tr>
<tr>
<td><code>conjugate()</code></td>
<td>Returns the group formed by conjugating <code>self</code> with <code>g</code>.</td>
</tr>
<tr>
<td><code>construction()</code></td>
<td>Return the construction of <code>self</code>.</td>
</tr>
<tr>
<td><code>cosets()</code></td>
<td>Returns a list of the cosets of <code>S</code> in <code>self</code>.</td>
</tr>
<tr>
<td><code>degree()</code></td>
<td>Returns the degree of this permutation group.</td>
</tr>
<tr>
<td><code>derived_series()</code></td>
<td>Wraps GAP’s <code>DirectProduct</code>, <code>Embedding</code>, and <code>Projection</code>.</td>
</tr>
<tr>
<td><code>domain()</code></td>
<td>Returns the underlying set that this permutation group acts on.</td>
</tr>
<tr>
<td><code>exponent()</code></td>
<td>Computes the exponent of the group.</td>
</tr>
<tr>
<td><code>fitting_subgroup()</code></td>
<td>Returns the Fitting subgroup of <code>self</code>.</td>
</tr>
<tr>
<td><code>fixed_points()</code></td>
<td>Return the list of points fixed by <code>self</code>, i.e., the subset of <code>.domain()</code> not moved by any element of <code>self</code>.</td>
</tr>
<tr>
<td><code>frattini_subgroup()</code></td>
<td>Returns the Frattini subgroup of <code>self</code>.</td>
</tr>
<tr>
<td><code>gen()</code></td>
<td>Returns the <code>i</code>-th generator of <code>self</code>; that is, the <code>i</code>-th element of the list <code>self</code>. <code>gens()</code>.</td>
</tr>
<tr>
<td><code>gens()</code></td>
<td>Return tuple of generators of this group. These need not be minimal, as they are the generators used in defining this group.</td>
</tr>
<tr>
<td><code>gens_small()</code></td>
<td>For this group, returns a generating set which has few elements. As neither irredundancy nor minimal length is proven, it is fast.</td>
</tr>
<tr>
<td><code>group_id()</code></td>
<td>Return the ID code of this group, which is a list of two integers.</td>
</tr>
<tr>
<td><code>group_primitive_id()</code></td>
<td>Return the index of this group in the GAP database of primitive groups.</td>
</tr>
<tr>
<td><code>has_element()</code></td>
<td>Returns boolean value of <code>item in self</code> - however ignores parentage.</td>
</tr>
<tr>
<td><code>holomorph()</code></td>
<td>The holomorph of a group as a permutation group.</td>
</tr>
<tr>
<td><code>homology()</code></td>
<td>Computes the group homology $H_n(G,F)$, where $F = Z$ if $p = 0$ and $F = Z/pZ$ if $p &gt; 0$ is a prime. Wraps HAP’s <code>GroupHomology</code> function, written by Graham Ellis.</td>
</tr>
<tr>
<td><code>homology_part()</code></td>
<td>Computes the $p$-part of the group homology $H_n(G,F)$, where $F = Z$ if $p = 0$ and $F = Z/pZ$ if $p &gt; 0$ is a prime. Wraps HAP’s <code>Homology</code> function, written by Graham Ellis, applied to the $p$-Sylow subgroup of $G$.</td>
</tr>
<tr>
<td><code>id()</code></td>
<td>(Same as <code>self.group_id()</code>) Return the ID code of this group, which is a list of two integers.</td>
</tr>
<tr>
<td><code>intersection()</code></td>
<td>Returns the permutation group that is the intersection of <code>self</code> and <code>other</code>.</td>
</tr>
<tr>
<td><code>irreducible_characters()</code></td>
<td>Returns a list of the irreducible characters of <code>self</code>.</td>
</tr>
<tr>
<td><code>is_cyclic()</code></td>
<td>Return True if this group is cyclic.</td>
</tr>
<tr>
<td><code>is_elementary_abelian()</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>is_isomorphic()</code></td>
<td>Return True if the groups are isomorphic.</td>
</tr>
<tr>
<td><code>is_monomial()</code></td>
<td>Returns 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.</td>
</tr>
<tr>
<td><code>is_nilpotent()</code></td>
<td>Return True if this group is nilpotent.</td>
</tr>
<tr>
<td><code>is_normal()</code></td>
<td>Return True if this group is a normal subgroup of <code>other</code>.</td>
</tr>
<tr>
<td><code>is_perfect()</code></td>
<td>Return True if this group is perfect. A group is perfect if it equals its derived subgroup.</td>
</tr>
<tr>
<td><code>is_pgroup()</code></td>
<td>Returns 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 $n$.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>is_polycyclic()</td>
<td>Return True if this group is polycyclic. A group is polycyclic if it has a</td>
</tr>
<tr>
<td></td>
<td>normal series with cyclic factors. (For finite groups, this is the same as</td>
</tr>
<tr>
<td></td>
<td>if the group is solvable - see is_solvable.)</td>
</tr>
<tr>
<td>is_primitive()</td>
<td>Returns True if self acts primitively on domain. A group $G$ acts primitively</td>
</tr>
<tr>
<td></td>
<td>on a set $S$ if</td>
</tr>
<tr>
<td>is_regular()</td>
<td>Returns True if self acts regularly on domain. A group $G$ acts regularly on</td>
</tr>
<tr>
<td></td>
<td>a set $S$ if</td>
</tr>
<tr>
<td>is_semi_regular()</td>
<td>Returns True if self acts semi-regularly on domain. A group $G$ acts semi-</td>
</tr>
<tr>
<td></td>
<td>regularly on a set $S$ if the point stabilizers of $S$ in $G$ are trivial.</td>
</tr>
<tr>
<td>is_simple()</td>
<td>Returns True if the group is simple. A group is simple if it has no proper</td>
</tr>
<tr>
<td></td>
<td>normal subgroups.</td>
</tr>
<tr>
<td>is_solvable()</td>
<td>Returns True if the group is solvable.</td>
</tr>
<tr>
<td>is_subgroup()</td>
<td>Returns True if self is a subgroup of other.</td>
</tr>
<tr>
<td>is_supersolvable()</td>
<td>Returns True if the group is supersolvable. A finite group is supersolvable</td>
</tr>
<tr>
<td></td>
<td>if it has a normal series with cyclic factors.</td>
</tr>
<tr>
<td>is_transitive()</td>
<td>Returns True if self acts transitively on domain. A group $G$ acts transitively</td>
</tr>
<tr>
<td></td>
<td>on set $S$ if for all $x, y \in S$ there is some $g \in G$ such that $x^g = y$.</td>
</tr>
<tr>
<td>isomorphism_to()</td>
<td>Return an isomorphism from self to right if the groups are isomorphic, oth-</td>
</tr>
<tr>
<td></td>
<td>erwise None.</td>
</tr>
<tr>
<td>isomorphism_type_info_simple_group()</td>
<td>If the group is simple, then this returns the name of the group.</td>
</tr>
<tr>
<td>iteration()</td>
<td>Return an iterator over the elements of this group.</td>
</tr>
<tr>
<td>largest_moved_point()</td>
<td>Return the largest point moved by a permutation in this group.</td>
</tr>
<tr>
<td>list()</td>
<td>Return list of all elements of this group.</td>
</tr>
<tr>
<td>lower_central_series()</td>
<td>Return the lower central series of this group as a list of permutation</td>
</tr>
<tr>
<td></td>
<td>groups.</td>
</tr>
<tr>
<td>minimal_generating_set()</td>
<td>Return a minimal generating set</td>
</tr>
<tr>
<td>molien_series()</td>
<td>Return the Molien series of a permutation group. The function</td>
</tr>
<tr>
<td>ngens()</td>
<td>Return the number of generators of self.</td>
</tr>
<tr>
<td>non_fixed_points()</td>
<td>Return the list of points not fixed by self, i.e., the subset of self.domain() moved by some element of self.</td>
</tr>
<tr>
<td>normal_subgroups()</td>
<td>Return the normal subgroups of this group as a (sorted in increasing order) list of permutation groups.</td>
</tr>
<tr>
<td>normalizer()</td>
<td>Returns the normalizer of $g$ in self.</td>
</tr>
<tr>
<td>normalizes()</td>
<td>Returns True if the group other is normalized by self. Wraps GAP’s IsNormal function.</td>
</tr>
<tr>
<td>poincare_series()</td>
<td>Return the Poincaré series of $G \mod p$ ($p \geq 2$ must be a prime), for $n$ large.</td>
</tr>
<tr>
<td>random_element()</td>
<td>Return a random element of this group.</td>
</tr>
<tr>
<td>representative_action()</td>
<td>Return an element of self that maps $x$ to $y$ if it exists.</td>
</tr>
<tr>
<td>semidirect_product()</td>
<td>The semidirect product of self with $N$.</td>
</tr>
<tr>
<td>sign_representation()</td>
<td>Return the sign representation of self over base_ring.</td>
</tr>
<tr>
<td>socle()</td>
<td>Returns the socle of self. The socle of a group $G$ is the subgroup gener-</td>
</tr>
<tr>
<td></td>
<td>ated by all minimal normal subgroups.</td>
</tr>
<tr>
<td>solvable_radical()</td>
<td>Returns the solvable radical of self. The solvable radical (or just radical)</td>
</tr>
<tr>
<td></td>
<td>of a group $G$ is the largest solvable normal subgroup of $G$.</td>
</tr>
<tr>
<td>stabilizer()</td>
<td>Return the subgroup of self which stabilize the given position. self and its</td>
</tr>
<tr>
<td></td>
<td>stabilizers must have same degree.</td>
</tr>
<tr>
<td>strong_generating_system()</td>
<td>Return a Strong Generating System of self according the given base for the right action of self on itself.</td>
</tr>
<tr>
<td>structure_description()</td>
<td>Return a string that tries to describe the structure of $G$.</td>
</tr>
<tr>
<td>subgroup()</td>
<td>Wraps the PermutationGroup_subgroup constructor. The argument gens is a list of elements of self.</td>
</tr>
<tr>
<td>subgroups()</td>
<td>Returns a list of all subgroups of self.</td>
</tr>
</tbody>
</table>
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sylow_subgroup()</code></td>
<td>Returns 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 $p$.</td>
</tr>
<tr>
<td><code>transversals()</code></td>
<td>If $G$ is a permutation group acting on the set $X = {1, 2, \ldots, n}$ and $H$ is the stabilizer subgroup of <code>&lt;integer&gt;</code>, 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 <code>self</code> by the stabilizer of <code>self</code> on <code>&lt;integer&gt;</code> position.</td>
</tr>
<tr>
<td><code>trivial_character()</code></td>
<td>Returns the trivial character of <code>self</code>.</td>
</tr>
<tr>
<td><code>upper_central_series()</code></td>
<td>Return the upper central series of <code>self</code> as a list of permutation groups.</td>
</tr>
</tbody>
</table>

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.
- 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 (Trac #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 trac ticket #26750)

REFERENCES:


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:

```python
sage: G = PermutationGroup([(1,2,3),(4,5)],[(3,4)])
sage: G
Permutation Group with generators [(3,4), (1,2,3)(4,5)]
```

We can also make permutation groups from PARI groups:

```python
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)]
```

We can also create permutation groups whose generators are Gap permutation objects:
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:

```python
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')]
```

There is an underlying gap object that implements each permutation group:

```python
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G._gap_()
Group( [ (1,2,3), (4,5) ] )
sage: gap(G)
Group( [ (1,2,3), (4,5) ] )
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()
sage: G1
Group( [ (3,4), (1,2,3)(4,5) ] )
sage: G2.GeneratorsSmallest()
[ (3,4,5), (2,3)(4,5), (1,2)(4,5) ]
```

```python
class sage.groups.perm_gps.permgroup.PermutationGroup_generic(gens=None, gap_group=None, canonicalize=True, domain=None, category=None)

Bases: sage.groups.group.FiniteGroup

A generic permutation group.

EXAMPLES:

```python
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()
```

Element
alias of `sage.groups.perm_gps.permgroup_element.PermutationGroupElement`

**Subgroup**
alias of `PermutationGroup_subgroup`

**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:**

```python
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 >
```

GAP algorithm is not guaranteed to produce minimal or canonical presentation:

```python
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, c | a^2, b^2, c^2, (c*b)^2, (c*a)^2, (b*a)^2 >
sage: K.as_permutation_group().is_isomorphic(DihedralGroup(5))
True
```

We can attempt to reduce the output presentation:

```python
sage: PermutationGroup(['(1,2,3,4,5)', '(1,3,5,2,4)']).as_finitely_presented_group()
Finitely presented group < a, b | b^2*a^2, b*a^-3*b*a^2 >
sage: PermutationGroup(['(1,2,3,4,5)', '(1,3,5,2,4)']).as_finitely_presented_group(reduced=True)
Finitely presented group < a | a^5 >
```

**AUTHORS:**

- Davis Shurbert (2013-06-21): initial version

**base**(seed=None)

Returns 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.
The argument seed is optional and must be a subset of the domain of 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]
```

blocks_all(representatives=True)

Returns 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()
```
sage: g = graphs.DodecahedralGraph()
sage: g.is_vertex_transitive()
True
sage: ag = g.automorphism_group()
sage: ag.is_primitive()
False
```

Computing its blocks representatives:

```
sage: ag.blocks_all()
[[0, 15]]
```

Now the full block:

```
sage: sorted(ag.blocks_all(representatives = False)[0])
[[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: G.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
```

cardinality is just an alias:

```
sage: PermutationGroup([(1,2,3)]).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)])
```

centralizer(g)

Returns 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 [(2,4), (1,3)] of (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
sage: g = G([(1,2,3,4)])
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)])

character(values)
Returns 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)
Character of Alternating group of order 4!/2 as a permutation group

ccharacter_table()
Returns 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)], [(1,2,3,4)]])
sage: G.order()
12
sage: G.character_table()

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-zeta3 - 1</td>
<td>zeta3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>zeta3 -zeta3 - 1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: G.character_table()

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1 -1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-1 1 -1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1 -1 -1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Type print(gap.eval("Display(%s)"%CT.name())) to display this nicely.

sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: G.order()
8
sage: G.character_table()

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1 -1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-1 1 -1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1 -1 -1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Again, type \texttt{print(gap.eval("Display(%s)"%CT.name()))} to display this nicely.

\begin{verbatim}
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), (5, 1, 2, -1, -1, 0, 0), (5, 1, -1, 2, -1, 0, 0), (8, 0, -1, -1, 0, 0, 0), (9, 1, 0, 0, 1, -1, -1), (10, -2, 1, 1, 0, 0, 0)]
\end{verbatim}

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 \texttt{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 \texttt{character_table_values(G)} $[v_1, \ldots, v_n] = [c_1, \ldots, c_n]$, where $[v_1, \ldots, v_n]$ = \texttt{character_table_values(G)} $(-1)[c_1, \ldots, c_n]$.

\textbf{AUTHORS:}

- David Joyner and William Stein (2006-01-04)

\textbf{cohomology}(n, p=0)

Computes the group cohomology $H^n(G, F)$, where $F = \mathbb{Z}$ if $p = 0$ and $F = \mathbb{Z}/p\mathbb{Z}$ if $p > 0$ is a prime.

Wraps HAP’s \texttt{GroupHomology} function, written by Graham Ellis.

\textbf{REQUIRES:} GAP package HAP (in gap_packages-*).spkg).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: G = SymmetricGroup(4)
sage: G.cohomology(1,2) # optional - gap_packages
Multiplicative Abelian group isomorphic to C2
sage: G = SymmetricGroup(3)
sage: G.cohomology(5) # optional - gap_packages
Trivial Abelian group
sage: G.cohomology(5,2) # optional - gap_packages
Multiplicative Abelian group isomorphic to C2
sage: G.homology(5,3) # optional - gap_packages
\end{verbatim}
Trivial Abelian group

sage: G.homology(5,4)                 # optional - gap_packages
Traceback (most recent call last):
...
ValueError: p must be 0 or prime

This computes $H^4(S_3, \mathbb{Z})$ and $H^4(S_3, \mathbb{Z}/2\mathbb{Z})$, respectively.

AUTHORS:
- David Joyner and Graham Ellis

REFERENCES:

cohomology_part(n, p=0)
Compute the p-part of the group cohomology $H^n(G, F)$, where $F = \mathbb{Z}$ if $p = 0$ and $F = \mathbb{Z}/p\mathbb{Z}$ 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_packages
Multiplicative Abelian group isomorphic to C2 x C2 x C2
sage: G = SymmetricGroup(3)
sage: G.cohomology_part(2,3)                 # optional - gap_packages
Multiplicative Abelian group isomorphic to C3

AUTHORS:
- David Joyner and Graham Ellis

commutator(\text{other}=\text{None})
Returns the commutator subgroup of a group, or of a pair of groups.

INPUT:
- \text{other} - default: \text{None} - a permutation group.

OUTPUT:

Let $G$ denote \text{self}. If \text{other} is \text{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 \text{other}, in the case that it is not \text{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
```

An abelian group will have a trivial commutator.

```
sage: G = CyclicPermutationGroup(10)
sage: 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
```

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
```

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
```
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
```

**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 withTelegram
˓→generators [(3,4), (1,2,3)(4,5)]), 
  Subgroup generated by [(1,3,5), (1,5)(3,4), (1,5)(2,4)] 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)])
```

**conjugacy_class()**

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
```

**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, ]
```

(continues on next page)
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()

Returns 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
```

```
sage: G = SymmetricGroup(5)
sage: G.conjugacy_classes_representatives()
[((), (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()
[((), ('a','b')), ('a','b','c')]
```

**AUTHORS:**

- David Joyner and William Stein (2006-01-04)

conjugacy_classes_subgroups()

Returns 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()
sage: 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 [(1,2)(3,4), (1,4)(2,3)] 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 [(1,2,3,4), (2)] of (Permutation Group with generators [(1,2)(3,4), (1,2,3,4))],
Subgroup generated by [(1,2)(3,4), (1,4)(2,3)] of (Permutation Group with generators [(1,2)(3,4), (1,2,3,4))]
```

Chapter 24. Permutation Groups
AUTHORS:

• David Joyner (2006-10)

conjugate(g)

Returns 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 [(1,4)(2,6)(3,5), (1,5,6,2,3,4)]
```

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 [(1,4)(2,6)(3,5), (1,5,6,2,3,4)]
sage: G = DihedralGroup(6)
sage: lst = [2,3,4,1]
sage: G.conjugate(lst)
Permutation Group with generators [(1,4)(2,6)(3,5), (1,5,6,2,3,4)]
sage: G = DihedralGroup(6)
sage: cycles = [(1,2,3,4)]
sage: G.conjugate(cycles)
Permutation Group with generators [(1,4)(2,6)(3,5), (1,5,6,2,3,4)]
```

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

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

The conjugating element is checked.

sage: G = SymmetricGroup(3)
sage: 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

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}

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)]

```
cosets(S, side='right')
```

Returns a list of the cosets of \( S \) in \( \text{self} \).

INPUT:

- \( S \) - a subgroup of \( \text{self} \). An error is raised if \( S \) is not a subgroup.
- \( \text{side} \) - default: ‘right’ - determines if right cosets or left cosets are returned. \( \text{side} \) 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 \( \text{self} \)) 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 the ordering of elements of \( \text{self} \)).

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)],
```

(continues on next page)
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.

```python
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)],
[(2,4,3), (1,4,3), (1,2)(3,4)],
[(1,2,4), (1,4,2), (1,3,4)]
sage: lc = A.cosets(stabilizer, side='left'); lc
[[()], (1,2,3), (1,3,2)],
[(2,4,3), (1,2)(3,4), (1,3,4)],
[(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
```

AUTHOR:
• Rob Beezer (2011-01-31)

```
degree()
```
Returns the degree of this permutation group.

EXAMPLES:
\begin{verbatim}
sage: S = SymmetricGroup(['a', 'b', 'c'])
sage: S.degree()
3
sage: G = PermutationGroup([(1,3),(4,5)])
sage: G.degree()
5

Note that you can explicitly specify the domain to get a permutation group of smaller degree:

\begin{verbatim}
sage: G = PermutationGroup([(1,3),(4,5)], domain=[1,3,4,5])
sage: G.degree()
4
\end{verbatim}
\end{verbatim}

\textbf{derived\_series()}

Return the derived series of this group as a list of permutation groups.

\textbf{EXAMPLES:}

These computations use pseudo-random numbers, so we set the seed for reproducible testing.

\begin{verbatim}
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 [(1,3,5), (1,5)(3,4), (1,5)(2,4)] of (Permutation Group
with generators [(3,4), (1,2,3)(4,5)])]
\end{verbatim}

\textbf{direct\_product(\textit{other}, maps=True)}

Wraps GAP's \texttt{DirectProduct}, \texttt{Embedding}, and \texttt{Projection}.

Sage calls GAP's \texttt{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 \texttt{Embedding}(\(D\), \textit{i}) returns the homomorphism embedding the \textit{i}-th factor into \(D\). The GAP operation \texttt{Projection}(\(D\), \textit{i}) gives the projection of \(D\) onto the \textit{i}-th factor. This method returns a 5-tuple: a permutation group and 4 morphisms.

\textbf{INPUT:}

\begin{itemize}
  \item \textit{self}, \textit{other} - permutation groups
\end{itemize}

\textbf{OUTPUT:}

\begin{itemize}
  \item \(D\) - a direct product of the inputs, returned as a permutation group as well
  \item \texttt{iota1} - an embedding of \textit{self} into \(D\)
  \item \texttt{iota2} - an embedding of \textit{other} into \(D\)
  \item \texttt{pr1} - the projection of \(D\) onto \textit{self} (giving a splitting 1 - \textit{other} - \(D\) - \textit{self} - 1)
  \item \texttt{pr2} - the projection of \(D\) onto \textit{other} (giving a splitting 1 - \textit{self} - \(D\) - \textit{other} - 1)
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: G = CyclicPermutationGroup(4)
sage: D = G.direct_product(G, False)
sage: D
Permutation Group with generators [(5,6,7,8), (1,2,3,4)]
\end{verbatim}
sage: D, iota1, iota2, pr1, pr2 = G.direct_product(G)
sage: D; iota1; iota2; pr1; pr2
Permutation Group with generators [(5,6,7,8), (1,2,3,4)]
Permutation group morphism:
  From: Cyclic group of order 4 as a permutation group
  To: Permutation Group with generators [(5,6,7,8), (1,2,3,4)]
  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 [(5,6,7,8), (1,2,3,4)]
  Defn: Embedding( Group( [(1,2,3,4), (5,6,7,8)] ), 2 )
Permutation group morphism:
  From: Permutation Group with generators [(5,6,7,8), (1,2,3,4)]
  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 [(5,6,7,8), (1,2,3,4)]
  To: Cyclic group of order 4 as a permutation group
  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)

**domain()**

Returns the underlying set that this permutation group acts on.

**EXAMPLES:**

```python
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'}
```

**exponent()**

Computes 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:**

```python
sage: G = AlternatingGroup(4)
sage: G.exponent()
6
```

**fitting_subgroup()**

Returns the Fitting subgroup of self.
The Fitting subgroup of a group $G$ is the largest nilpotent normal subgroup of $G$.

**EXAMPLES:**

```python
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)])
```

**fixed_points()**

Return the list of points fixed by `self`, i.e., the subset of `.domain()` not moved by any element of `self`.

**EXAMPLES:**

```python
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]
```

**frattini_subgroup()**

Returns the Frattini subgroup of `self`.

The Frattini subgroup of a group $G$ is the intersection of all maximal subgroups of $G$.

**EXAMPLES:**

```python
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)
```

**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:**

```python
sage: P8=PSp(8,3)
sage: P8.gap()
(continues on next page)```
<permutation group of size 65784756654489600 with 2 generators>
sage: gap(P8) == P8.gap()
False
sage: S3 = SymmetricGroup(3)
sage: S3.gap()
Sym( [ 1 .. 3 ] )
sage: gap(S3) == S3.gap()
False

\textbf{gen}(\texttt{i=None})

Returns the \texttt{i}-th generator of \texttt{self}; that is, the \texttt{i}-th element of the list \texttt{self.gens()}.

The argument \texttt{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:

\begin{verbatim}
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)
\end{verbatim}

\textbf{gens}()

Return tuple of generators of this group. These need not be minimal, as they are the generators used in defining this group.

**EXAMPLES:**

\begin{verbatim}
sage: G = PermutationGroup([[\( (1,2,3) \)], \((1,2)\)])
sage: G.gens()
[(1,2), (1,2,3)]
\end{verbatim}

Note that the generators need not be minimal, though duplicates are removed:

\begin{verbatim}
sage: G = PermutationGroup([[\( (1,2) \)], \((1,3)\)], [[\( 2,3) \)], [[\( 1,2) \)])
sage: G.gens()
[(2,3), (1,2), (1,3)]
\end{verbatim}

We can use index notation to access the generators returned by \texttt{self.gens}:
sage: g[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:**

```python
    # 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
```

The output may be unpredictable, due to the use of randomized algorithms in GAP. Note that both the
following answers are equally valid.

```python
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
True
```

**group_id()**

Return the ID code of this group, which is a list of two integers.

**EXAMPLES:**

```python
sage: G = PermutationGroup([[(1,2,3),(4,5)],[1,2]])
sage: 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:**

```python
sage: G = PermutationGroup([[(1,2,3,4,5)], [(1,5),(2,4)]]
sage: G.group_primitive_id()
(continues on next page)```
From the information of the degree and the identification number, you can recover the isomorphism class of your group in the GAP database:

```sage
sage: H = PrimitiveGroup(5,2)
sage: G == H
False
sage: G.is_isomorphic(H)
True
```

**has_element(item)**

Returns boolean value of item in self - however ignores parentage.

**EXAMPLES:**

```sage
sage: G = CyclicPermutationGroup(4)
sage: gens = G.gens()
sage: H = DihedralGroup(4)
sage: g = G([(1,2,3,4)]); g
(1,2,3,4)
sage: G.has_element(g)
True
sage: h = H([(1,2),(3,4)]); h
(1,2)(3,4)
sage: G.has_element(h)
False
```

**has_regular_subgroup(return_group=False)**

Return whether the group contains a regular subgroup.

**INPUT:**

- `return_group` (boolean) – If `return_group = 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
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)]
```

But the automorphism group of Petersen’s graph does not:

```sage
sage: G = graphs.PetersenGraph().automorphism_group()
sage: 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:

Returns 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)]
```

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()
[(3,8)(4,7), (2,3,5,8), (2,5)(3,8), (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
```

Author:

- Kevin Halasz (2012-08-14)

homology($n$, $p=0$)

Computes the group homology $H_n(G,F)$, where $F = \mathbb{Z}$ if $p = 0$ and $F = \mathbb{Z}/p\mathbb{Z}$ 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 $H_7(S_5, \mathbb{Z})$, $H_7(S_5, \mathbb{Z}/2\mathbb{Z})$, $H_7(S_5, \mathbb{Z}/3\mathbb{Z})$, and $H_7(S_5, \mathbb{Z}/5\mathbb{Z})$, respectively.

To compute the 2-part of $H_7(S_5, \mathbb{Z})$, use the homology_part function.

EXAMPLES:

```python
sage: G = SymmetricGroup(5)
sage: G.homology(7) # optional - gap_packages
Multiplicative Abelian group isomorphic to C2 x C2 x C4 x C3 x C5
sage: G.homology(7,2) # optional - gap_packages
Multiplicative Abelian group isomorphic to C2 x C2 x C2 x C2 x C2
sage: G.homology(7,3) # optional - gap_packages
Multiplicative Abelian group isomorphic to C3
sage: G.homology(7,5) # optional - gap_packages
Multiplicative Abelian group isomorphic to C5
```

REFERENCES:


• D. Joyner, “A primer on computational group homology and cohomology”, http://front.math.ucdavis.edu/0706.0549

`homology_part(n, p=0)`

Computes the $p$-part of the group homology $H_n(G, F)$, where $F = \mathbb{Z}$ if $p = 0$ and $F = \mathbb{Z}/p\mathbb{Z}$ 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:

```python
sage: G = SymmetricGroup(5)
sage: G.homology_part(7,2) # optional - gap_packages
Multiplicative Abelian group isomorphic to C2 x C2 x C2 x C2 x C4
```

AUTHORS:

• David Joyner and Graham Ellis

`id()`

(As same as self.group_id().) Return the ID code of this group, which is a list of two integers.

EXAMPLES:

```python
sage: G = PermutationGroup([(1,2,3),(4,5)], [(1,2)])
sage: 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()
Sage: e
()
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()
()

intersection(other)
Returns 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)
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 [()]

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

The group other is verified as such.

Sage: H = DihedralGroup(4)
Sage: H.intersection('junk')
irreducible_characters()

Returns a list of the irreducible characters of self.

EXAMPLES:

```python
sage: irr = SymmetricGroup(3).irreducible_characters()
sage: [x.values() for x in irr]
[[1, -1, 1], [2, 0, -1], [1, 1, 1]]
```

is_abelian()

Return True if this group is abelian.

EXAMPLES:

```python
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
```

is_commutative()

Return True if this group is commutative.

EXAMPLES:

```python
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)'])
sage: G.is_commutative()
True
```

is_cyclic()

Return True if this group is cyclic.

EXAMPLES:

```python
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
```

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:
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:
```python
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
```

is_monomial()
Returns 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:
```python
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_monomial()
True
```

is_nilpotent()
Return True if this group is nilpotent.

EXAMPLES:
```python
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_nilpotent()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_nilpotent()
True
```

is_normal(other)
Return True if this group is a normal subgroup of other.

EXAMPLES:
is_normal()

Return True if this group is normal. A group is normal if it equals its derived subgroup. EXAMPLES:

```
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
```

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
```

is_pgroup()

Returns 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
```

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
```

is_primitive(domain=None)

Returns 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 \texttt{self} on its domain:

\begin{verbatim}
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
\end{verbatim}

You can specify a domain on which to test primitivity:

\begin{verbatim}
sage: G = PermutationGroup([[1,2,3,4],[2,4]])
sage: G.is_primitive([1..4])
    False
sage: G.is_primitive([1,2,3])
    True
sage: G = PermutationGroup([[3,4,5,6],[3,4]]) #S_4 on [3..6]
sage: G.is_primitive(G.non_fixed_points())
    True
\end{verbatim}

\texttt{is_regular}(domain=None)

Returns True if \texttt{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:

\begin{verbatim}
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
\end{verbatim}

You can pass in a domain on which to test regularity:

\begin{verbatim}
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
\end{verbatim}

\texttt{is_semi_regular}(domain=None)

Returns True if \texttt{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:

\begin{verbatim}
sage: G = PermutationGroup([[1,2,3,4]])
sage: G.is_semi_regular()
    True
sage: G = PermutationGroup([[1,2,3,4],[5,6]])
\end{verbatim}
sage: 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

**is_simple()**

Returns 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

**is_solvable()**

Returns True if the group is solvable.

EXAMPLES:

sage: G = PermutationGroup(['(1,2,3)(4,5)'])

sage: G.is_solvable()
True

**is_subgroup(other)**

Returns True if self is a subgroup of other.

EXAMPLES:

sage: G = AlternatingGroup(5)
sage: H = SymmetricGroup(5)
sage: G.is_subgroup(H)
True

**is_supersolvable()**

Returns 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

**is_transitive(domain=None)**

Returns 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:
```python
sage: G = SymmetricGroup(5)
sage: G.is_transitive()
True
sage: G = PermutationGroup(['(1,2)(3,4)(5,6)'])
sage: G.is_transitive()
False

sage: G = PermutationGroup([[1,2,3,4,5]],[1,2]]) #S_5 on [1..5]
sage: G.is_transitive([1,4,5])
True
sage: G.is_transitive([2..6])
False
sage: G.is_transitive(G.non_fixed_points())
True

sage: H = PermutationGroup([[1,2,3]],[4,5,6]])
sage: H.is_transitive(H.non_fixed_points())
False
```
EXAMPLES:

```python
sage: G = CyclicPermutationGroup(5)
sage: 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:

```python
sage: G = PermutationGroup([[1,2], [2,3]])
```

```python
sage: list(G.iteration())
[(1,2,3), (1,3,2), (1,2), (1,3)]
```

```python
sage: list(G.iteration(algorithm="BFS"))
[(1,2,3), (1,3,2), (1,2), (1,3)]
```

```python
sage: list(G.iteration(algorithm="DFS"))
[(1,2,3), (1,2), (1,3), (1,2,3), (2,3)]
```

**largest_moved_point()**

Return the largest point moved by a permutation in this group.

EXAMPLES:

```python
sage: G = PermutationGroup([[1,2], [3,4]])
sage: G.largest_moved_point()
4
```

```python
sage: G = PermutationGroup([[1,2], [3,4], [1,2,3,4,10]])
sage: G.largest_moved_point()
10
```
```python
sage: G = PermutationGroup([[('a', 'b', 'c'), ('d', 'e')]])
sage: G.largestMovedPoint()
'e'
```

**Warning:** The name of this function is not good; this function should be deprecated in term of degree:

```python
sage: P = PermutationGroup([[1, 2, 3, 4]])
sage: P.largestMovedPoint()
4
sage: P.cardinality()
1
```

**list()**

Return list of all elements of this group.

**EXAMPLES:**

```python
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)]
```

```python
sage: G = PermutationGroup([['a', 'b']], domain=('a', 'b'))

sage: 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.

```python
sage: set_random_seed(0)
sage: G = PermutationGroup([[1, 2, 3], [(4, 5)]]

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 [(1,3,5), (1,5)(3,4), (1,5)(2,4)] of (Permutation Group with generators [(3,4), (1,2,3)(4,5)])]
```

**minimal_generating_set()**

Return a minimal generating set

**EXAMPLES:**

```python
sage: g = graphs.CompleteGraph(4)
sage: g.relabel(['a', 'b', 'c', 'd'])
sage: mgs = g.automorphismGroup().minimalGeneratingSet(); len(mgs)
2
```

(continues on next page)
molien_series()

Return the Molien series of a permutation group. The function

\[ M(x) = \frac{1}{|G|} \sum_{g \in G} \det(1 - x * g)^{-1} \]

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
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)
```

Some further tests (after trac ticket #15817):

```sage
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
```

This works for not-transitive groups:

```sage
sage: G = PermutationGroup([[1,2],[3,4]])
sage: G.molien_series() / S4ms
x^4 + x^3 + 2*x^2 + x + 1
```

This works for groups with fixed points:

```sage
sage: G = PermutationGroup([[2,]])
sage: G.molien_series()
1/(x^2 - 2*x + 1)
```

ngens()

Return the number of generators of \( \text{self} \).

EXAMPLES:

```sage
sage: A4 = PermutationGroup([[1,2,3], [2,3,4]]); A4
Permutation Group with generators [(2,3,4), (1,2,3)]
sage: A4.ngens()
2
```

non_fixed_points()

Return the list of points not fixed by \( \text{self} \), i.e., the subset of \( \text{self}.\text{domain()} \) moved by some element of \( \text{self} \).
EXAMPLES:

```python
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]
```

**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:

```python
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
```

**normalizer(g)**

Returns the normalizer of $g$ in self.

EXAMPLES:

```python
sage: G = PermutationGroup([[(1,2),(3,4)]]
sage: g = G([(1,3)])
sage: G.normalizer(g)
Subgroup generated by [(2,4), (1,3)] of (Permutation Group with generators [(1, ˓→2),(3,4)])
sage: g = G([(1,2,3,4)])
sage: G.normalizer(g)
Subgroup generated by [(2,4), (1,2,3,4), (1,3)(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 [(2,4), (1,2,3,4), (1,3)(2,4)] of (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)])
```

**normalizes(other)**

Returns 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:

```python
dsage: G = PermutationGroup(['(1,2,3)(4,5)']
dsage: H = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
```
In the last example, $G$ and $H$ are disjoint, so each normalizes the other.

one()

Return the identity element of this group.

EXAMPLES:

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:
(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)] ])
('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})
```

On tuples:

```
sage: S3.orbit((1,2), 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
```

Action of $S_4$ (on a nonstandard domain) on tuples of sets:

```
sage: S4 = PermutationGroup([ [(c, d)], [(a, c)], [(a, b)] ])
sage: S4.orbit((( 'a', 'c'), ('b', 'd') ),"OnTuplesSets") # py2
((( 'a', 'c'), { 'b', 'd' } ),
 ( ( 'a', 'd' ), { 'c', 'b' } ),
 ( 'c', 'b' ), { 'a', 'd' } ),
 ( { 'b', 'd' }, { 'a', 'c' } ),
 ( { 'c', 'd' }, { 'a', 'b' } ),
 ( { 'a', 'b' }, { 'c', 'd' } ))
```

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: S4.orbit((( 11,(12,13)), (12,(12,11))),('b','d') )"OnTuplesSets") # py2
(((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' } ))
```
orbits()

Returns the orbits of the elements of the domain under the default group action.

EXAMPLES:

```python
sage: G = PermutationGroup([[(3,4)], [(1,3)]]
```

```python
sage: G.orbits()
[[1, 3, 4], [2]]
```

```python
sage: G = PermutationGroup([(1,2),(3,4)], [(1,2,3,4,10)])
```

```python
sage: G.orbits()
[[1, 2, 3, 4, 10], [5], [6], [7], [8], [9]]
```

```python
sage: G = PermutationGroup(["c", "d"], ["a", "c"], ["b"], ["b"])
```

```python
sage: G.orbits()
[['a', 'c', 'd'], ['b']]
```

The answer is cached:

```python
sage: G.orbits() is G.orbits()
True
```

AUTHORS:

• Nathan Dunfield

order()

Return the number of elements of this group. See also: G.degree()

EXAMPLES:

```python
sage: G = PermutationGroup([(1,2,3),(4,5)], [(1,2)])
```

```python
sage: G.order()
12
```

```python
sage: G = PermutationGroup([()])
```

```python
sage: G.order()
1
```

cardinality is just an alias:

```python
sage: PermutationGroup([(1,2,3)]).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 $H_k(G, \mathbb{Z}/p\mathbb{Z})$, for all $k$ in the range $1 \leq k \leq n$.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

EXAMPLES:

```python
sage: G = SymmetricGroup(5)
```

```python
sage: G.poincare_series(2,10) # optional - gap_packages
```

(continues on next page)
AUTHORS:

• David Joyner and Graham Ellis

quotient(N, **kwds)
Returns 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 [()]
```

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
()```

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```

24.3. Permutation groups
**semidirect_product**(\(N, \text{mapping}, \text{check=\text{True}}\))

The semidirect product of \(\text{self}\) with \(N\).

**INPUT:**

- \(N\) - A group which is acted on by \(\text{self}\) and naturally embeds as a normal subgroup of the returned semidirect product.
- \(\text{mapping}\) - A pair of lists that together define a homomorphism, \(\phi : \text{self} \rightarrow \text{Aut}(N)\), by giving, in the second list, the images of the generators of \(\text{self}\) in the order given in the first list.
- \(\text{check}\) - A boolean that, if set to False, will skip the initial tests which are made on \(\text{mapping}\). 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 \(\text{self}\) and \(N\) defined by the action of \(\text{self}\) on \(N\) given in \(\text{mapping}\) (note that a homomorphism from \(A\) to the automorphism group of \(B\) is equivalent to an action of \(A\) on the \(B\)'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 \(\text{self}\) and \(N\) on a set of size \(|N|\). The generators of \(N\) are given as their right regular representations, while the generators of \(\text{self}\) are defined by the underlying action of \(\text{self}\) 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 \(\text{self}\) are placed on additional letters and adjoined to the output’s generators of \(\text{self}\).

**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.

```python
sage: C2 = CyclicPermutationGroup(2)
sage: C8 = CyclicPermutationGroup(8)
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)]
```

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.

```python
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

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)

AUTHOR:

• Kevin Halasz (2012-8-12)

sign_representation(base_ring=None, side='twosided')

Return the sign representation of self over base_ring.

INPUT:

• base_ring – (optional) the base ring; the default is \( \mathbb{Z} \)

• side – ignored

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

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)])
Note that this function uses the ordering from the domain:

```python
sage: S = SymmetricGroup(['a','b','c'])
sage: S.smallest Moved Point()
'a'
```

**socle()**

Returns the socle of `self`. The socle of a group \( G \) is the subgroup generated by all minimal normal subgroups.

**EXAMPLES:**

```python
sage: G=SymmetricGroup(4)
sage: G.socle()
Subgroup generated by [(1,2)(3,4), (1,4)(2,3)] of (Symmetric group of order 4! as a permutation group)
sage: G.socle().socle()
Subgroup generated by [(1,2)(3,4), (1,4)(2,3)] of (Subgroup generated by [(1,2)(3,4), (1,4)(2,3)] of (Symmetric group of order 4! as a permutation group))
```

**solvable_radical()**

Returns the solvable radical of `self`. The solvable radical (or just radical) of a group \( G \) is the largest solvable normal subgroup of \( G \).

**EXAMPLES:**

```python
sage: G=SymmetricGroup(4)
sage: G.solvable_radical()
Subgroup generated by [(1,2), (1,2,3,4)] 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)
```

**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:**

```python
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)])
```
The stabilizer of a set of points:

```python
sage: s10 = groups.permutation.Symmetric(10)
sage: s10.stabilizer([1..3], "OnSets").cardinality()
30240
sage: factorial(3)*factorial(7)
30240
```

```python
sage: G = PermutationGroup([[1,2,3,4,10]], domain='abcd')
sage: G.stabilizer('a')
Subgroup generated by [[c\',d']] of (Permutation Group with generators [[c\',d'], [a\',c']])
```

24.3. Permutation groups

(continues on next page)
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')])

\texttt{strong\_generating\_system}(\texttt{base\_of\_group=None, implementation='sage'})

Return a Strong Generating System of self according the right action of self on itself.

\texttt{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, \texttt{base\_of\_group} is \([1, 2, 3, \ldots, d]\) where \(d\) is the degree of the group.

For \texttt{base\_of\_group} = [\texttt{pos}_1, \texttt{pos}_2, \ldots, \texttt{pos}_d] let \(G_i\) be the subgroup of \(G = \text{self}\) which stabilizes \texttt{pos}_1, \texttt{pos}_2, \ldots, \texttt{pos}_i, so

\[
G = G_0 \supset G_1 \supset G_2 \supset \cdots \supset G_n = \{e\}
\]

Then the algorithm returns \([G_i, \text{transversals}(\texttt{pos}_i+1)]\) for \(1 \leq i \leq n\)

INPUT:

- \texttt{base\_of\_group} (optional) – (default: \([1, 2, 3, \ldots, d]\)) a list containing the integers 1, 2, \ldots, \(d\) in any order, where \(d\) is the degree of self
- \texttt{implementation} – (default: "sage") either
  - "sage" - use the direct implementation in Sage
  - "gap" - if used, the \texttt{base\_of\_group} must be \texttt{None} and the computation is directly performed in GAP

OUTPUT:

A list of lists of permutations from the group, which form a strong generating system.

\textbf{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 \texttt{base\_of\_group} is given in the first situation.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: G = PermutationGroup([[(7,8)],[(3,4)],[(4,5)]])
sage: G.strong_generating_system() [[[0], [0], [0], [0]], [0], [0], [0], [0], [0], [0], [0]]
sage: G = PermutationGroup([[(1,2,3)],[(1,2)]])
sage: G.strong_generating_system() [[[0], (1,2)(3,4), (1,3)(2,4), (1,4)(2,3)], [(0), (2,4), (2,3,4)], [0], [0], [0], [0]]
sage: G = PermutationGroup([[(1,2,3)],[4,5,7],[1,4,6]])
sage: G.strong_generating_system() [[[0], (1,2,3), (1,4,6), (1,3,2), (1,5,7,4,6), (1,6,4), (1,7,5,4,6)], [(0), (2,3,6), (2,6,3), (2,7,5,6,3), (2,5,6,3)(4,7), (2,4,5,6,3)]
\end{verbatim}


```plaintext
[[(), (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])
[[(), [()], [()], [()]]]
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)],
```

**structure_description**($G$, latex=False)

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

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

```
sage: D3 = DihedralGroup(3)
sage: D3.structure_description()
'S3'
```

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

```
sage: D4 = DihedralGroup(4)
sage: D4.structure_description()
'D4'
```

Works for finitely presented groups (trac ticket #17573):

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

And matrix groups (trac ticket #17573):

```
sage: groups.matrix.GL(4,2).structure_description()
'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 [(1,2,3), ....
˓→(1,2,3)])
```

```
subgroups()
```
Returns 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 [(2,3), (1,2,3)] of (Symmetric group of order 3! as a permutation group)]
sage: G = CyclicPermutationGroup(14)
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 [(1,2,3,4,5,6,7,8,9,10,11,12,13,14), (1,3,5,7,9,11,13)(2,4,6,8,10,12,14)] of (Cyclic group of order 14 as a permutation group)]
```

**AUTHOR:**
- Rob Beezer (2011-01-24)

**sylow_subgroup(p)**

Returns 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)
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)])
```

(continues on next page)
transversals(point)

If G is a permutation group acting on the set \( X = \{1, 2, \ldots, n\} \) and H is the stabilizer subgroup of \(<\text{integer}>\),
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 \(<\text{integer}>\)
position.

EXAMPLES:

```sage
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')]```

trivial_character()

Returns the trivial character of self.

EXAMPLES:

```sage
sage: SymmetricGroup(3).trivial_character()
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
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)])]```

class sage.groups.perm_gps.permgroup.PermutationGroup_subgroup(ambient, gens=None, gap_group=None, domain=None, category=None, canonicalize=True, check=True)

Bases: sage.groups.perm_gps.permgroup.PermutationGroup_generic

Subgroup subclass of PermutationGroup_generic, so instance methods are inherited.

EXAMPLES:

```sage
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)]

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

is_normal(\(\text{other}=\text{None}\))

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

sage.groups.perm_gps.permgroup.direct_product_permgroups(\(P\))

Takes 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

sage.groups.perm_gps.permgroup.from_gap_list(\(G, \text{src}\))

Convert a string giving a list of GAP permutations into a list of elements of G.
EXAMPLES:

```python
sage: from sage.groups.perm_gps.permgroup import from_gap_list
sage: G = PermutationGroup([(1,2,3),(4,5)], [(3,4)])
```

```python
sage: L = from_gap_list(G, '[(1,2,3)(4,5), (3,4)]'); L
[(1,2,3)(4,5), (3,4)]
```

```python
sage: L[0].parent() is G
True
sage: L[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:

```python
sage: from sage.groups.perm_gps.permgroup import hap_decorator
sage: def foo(self, n, p=0): print("Done")
```

```python
sage: foo = hap_decorator(foo)
```

```python
sage: foo(None, 3)
# optional - gap_packages
Done
```

```python
sage: foo(None, 3, 0)
# optional - gap_packages
Done
```

```python
sage: foo(None, 3, 5)
# optional - gap_packages
Done
```

```python
sage: foo(None, 3, 4)
#optional - gap_packages
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:

```python
sage: sage.groups.perm_gps.permgroup.load_hap() # optional - gap_packages
```

### 24.4 “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
- **PrimitiveGroup**, \(n^{th}\) **primitive group of degree** \(d\) from the GAP tables of primitive groups
- **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 \(GF(q)\)
- **PSL(n,q)**, **projective special linear group** of \(n \times n\) matrices over the finite field \(GF(q)\)
- **PSp(2n,q)**, **projective symplectic linear group** of \(2n \times 2n\) matrices over the finite field \(GF(q)\)
- **PSU(n,q)**, **projective special unitary group** of \(n \times n\) matrices having coefficients in the finite field \(GF(q^2)\) 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 \(GF(q^2)\) that respect a fixed nondegenerate sesquilinear form, modulo the centre.
- **SuzukiGroup**(q), Suzuki group over \(GF(q)\), \(2B_2(2^{2k+1}) = Sz(2^{2k+1})\).
- **ComplexReflectionGroup**, the complex reflection group \(G(m,p,n)\) or the exceptional complex reflection group \(G_m\)

**AUTHOR:**
- David Joyner (2007-06): split from permgp.py (suggested by Nick Alexander)

**REFERENCES:**

**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.

```python
class sage.groups.perm_gps.permgroup_named.AlternatingGroup(domain=None)
Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_symalt

The alternating group of order \(n!/2\), as a permutation group.

**INPUT:**
- \(n\) – a positive integer, or list or tuple thereof

**Note:** This group is also available via groups.permutation.Alternating().

**EXAMPLES:**
```
```python
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()
```

```python
class sage.groups.perm_gps.permgroup_named.ComplexReflectionGroup(m, p=None, n=None)
Bases: sage.groups.perm_gps.permgroup_named.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 $1 \leq i < n$,

$$s_n \mapsto s_n^{-1}s_{n-1}s_n, \quad s_{n+1} \mapsto s_n^p.$$  

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:**

```python
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()
```

(continues on next page)
Join of Category of finite enumerated permutation groups and Category of finite complex reflection groups

```python
sage: G = groups.permutation.ComplexReflection(3, 3, 4)
sage: G.cardinality()
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()
```

```python
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):
... Not Implemented: exceptiona 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 \(\ell\) 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:
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)
```

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
```

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)

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)}

class sage.groups.perm_gps.permgroup_named.CyclicPermutationGroup(n)
Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

A cyclic group of order n, as a permutation group.

INPUT:

n – a 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
as_AbelianGroup()
Returns the corresponding Abelian Group instance.

EXAMPLES:

```
sage: C = CyclicPermutationGroup(8)
sage: 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
```

is_commutative()
Return True if this group is commutative.

EXAMPLES:

```
sage: C = CyclicPermutationGroup(8)
sage: C.is_commutative()
True
```

class sage.groups.perm_gps.permgroup_named.DiCyclicGroup(n)
Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

The dicyclic group of order 4n, for n ≥ 2.

INPUT:

• n – a 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 \( 0 ≤ i < 2n, j = 0, 1 \). 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 \( \text{groups.permutation.DiCyclic()} \).

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
```

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
```

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)]
sage: 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
```

24.4. “Named” Permutation groups (such as the symmetric group, S_n)
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
```

**is_commutative()**

Return True if this group is commutative.

EXAMPLES:

```
sage: D = DiCyclicGroup(12)
sage: D.is_commutative()
False
```

class sage.groups.perm_gps.permgroup_named.DihedralGroup(n)

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

The Dihedral group of order 2\(n\) for any integer \(n \geq 1\).

INPUT:

- \(n\) – a positive integer

OUTPUT:

The dihedral group of order 2\(n\), 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), (1,4,5,3), (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()
[(1,2,3,4,5), (1,5)(2,4)]

sage: DihedralGroup(0)
Traceback (most recent call last):
...
ValueError: n must be positive

class sage.groups.perm_gps.permgroup_named.GeneralDihedralGroup(factors)
Bases: sage.groups.perm_gps.permgroup.PermutationGroup_generic
The Generalized Dihedral Group generated by the abelian group with direct factors in the input list.

INPUT:

- **factors** - a 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

\[ \langle a, b, c | a^3, b^3, c^2, ab = ba, ac = ca^{-1}, bc = cb^{-1} \rangle \]

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()

(continues on next page)
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.

```python
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
```

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.

```python
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
```

A cyclic input yields a Classical Dihedral Group.

```python
sage: G = GeneralDihedralGroup([6])
sage: D = DihedralGroup(6)
sage: 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).

```python
sage: G = GeneralDihedralGroup([6,18,33,60])
sage: (6*18*33*60)**2
427680
sage: G.order()
427680
sage: G.is_solvable()
True
```

(continues on next page)
AUTHOR:
• Kevin Halasz (2012-7-12)

class sage.groups.perm_gps.permgroup_named.JankoGroup(n)
Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

Janko Groups \( J_1, J_2, \) and \( J_3 \). (Note that \( J_4 \) is too big to be treated here.)

INPUT:
• \( n \) – an integer among \( \{1, 2, 3\} \).

EXAMPLES:

```
sage: G = groups.permutation.Janko(1); G
# optional - gap_packages internet
Janko group J1 of order 175560 as a permutation group
```

class sage.groups.perm_gps.permgroup_named.KleinFourGroup
Bases: sage.groups.perm_gps.permgroup_named.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)
[(1,2), (1,3), (1,4), (2,3)]
```

AUTHOR: – Bobby Moretti (2006-10)

class sage.groups.perm_gps.permgroup_named.MathieuGroup(n)
Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

The Mathieu group of degree \( n \).

INPUT:
n – a 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:
class sage.groups.perm_gps.permgroup_named.PGL(n, q, name='a')

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_plg

The projective general linear groups over GF(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 GF(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
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

class sage.groups.perm_gps.permgroup_named.PGU(n, q, name='a')

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_pug

The projective general unitary groups over GF(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 GF(q)

OUTPUT:

PGU(n,q)
Note: This group is also available via `groups.permutation.PGU()`.

EXAMPLES:

```python
sage: PGU(2,3)
The projective general unitary group of degree 2 over Finite Field of size 3
sage: G = PGU(2, 8, name='alpha'); G
The projective general unitary group of degree 2 over Finite Field in alpha of size 3^2
sage: G.base_ring()
Finite Field in alpha of size 3^2
```

class `sage.groups.perm_gps.permgroup_named.PSL(n, q, name='a')`

Bases: `sage.groups.perm_gps.permgroup_named.PermutationGroup_plg`

The projective special linear groups over GF(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 GF(q)

OUTPUT:

the group PSL(n,q)

Note: This group is also available via `groups.permutation.PSL()`.

EXAMPLES:

```python
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
```

We create two groups over nontrivial finite fields:

```python
sage: G = PSL(2, 4, 'b'); G
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
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 a of size 2^3
sage: G.category()
(continues on next page)
```
ramification_module_decomposition_hurwitz_curve()

Helps compute the decomposition of the ramification module for the Hurwitz curves X (over CC say) with automorphism group G = PSL(2,q), q a “Hurwitz prime” (ie, p is ±1 (mod 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()).


EXAMPLES:

```
sage: G = PSL(2,13)
sage: G.ramification_module_decomposition_hurwitz_curve() # random, optional
\[0, 7, 7, 12, 12, 12, 13, 15, 14\]
```

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()).


EXAMPLES:

```
sage: G = PSL(2,7)
sage: G.ramification_module_decomposition_modular_curve() # random, optional
\[0, 4, 3, 6, 7, 8\]
```

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 sage.groups.perm_gps.permgroup_named.PSp
```

class sage.groups.perm_gps.permgroup_named.PSU(n, q, name='a')

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_pug

The projective special unitary groups over GF(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 GF(q)

OUTPUT:

\(PSU(n,q)\)

**Note:** This group is also available via `groups.permutation.PSU()`.

**EXAMPLES:**

```
sage: PSU(2,3)
The projective special unitary group of degree 2 over Finite Field of size 3
sage: G = PSU(2, 8, name='alpha'); G
The projective special unitary group of degree 2 over Finite Field in alpha of size \(\rightarrow 2^3\)
sage: G.base_ring()
Finite Field in alpha of size \(2^3\)
```

class sage.groups.perm_gps.permgroup_named.PSp(n, q, name='a')

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_plg

The projective symplectic linear groups over GF(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 GF(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)]
```

(continues on next page)
sage: G.order()
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), (1,5,14,17,27,22,19,
    →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
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

class sage.groups.perm_gps.permgroup_named.PermutationGroup_plg(gens=None, gap_group=None,
canonicalize=True, domain=None, category=None)

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

base_ring()

EXAMPLES:

    sage: G = PGL(2,3)
sage: G.base_ring()
Finite Field of size 3

    sage: G = PSL(2,3)
sage: G.base_ring()
Finite Field of size 3

matrix_degree()

EXAMPLES:

    sage: G = PSL(2,3)
sage: 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: sage.groups.perm_gps.permgroup_named.PermutationGroup_plg

field_of_definition()

EXAMPLES:

    sage: PSU(2,3).field_of_definition()
    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: sage.groups.perm_gps.permgroup_named.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: sage.structure.unique_representation.CachedRepresentation, sage.groups.perm_gps.permgroup.PermutationGroup_generic

Todo: Fix the broken hash.

def test():
    sage: G = SymmetricGroup(6)
    sage: G3 = G.subgroup([(1,2,3,4,5,6), (1,2)])
    sage: hash(G) == hash(G3)  # todo: Should be True!
    False

class sage.groups.perm_gps.permgroup_named.PrimitiveGroup(d, n)

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

The primitive group from the GAP tables of primitive groups.

INPUT:

• d – non-negative 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:

def test():
    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

Warning: this follows GAP's naming convention of indexing the primitive groups starting from 1:

def test():
    sage: PrimitiveGroup(5,0)
    Traceback (most recent call last):
ValueError: Index n must be in \{1, \ldots, 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!
```

\textbf{group\_primitive\_id()}  
Return the index of this group in the GAP database of primitive groups.

\textbf{OUTPUT:}
A positive integer, following GAP’s conventions.

\textbf{EXAMPLES:}
```
sage: G = PrimitiveGroup(5,2); G.group_primitive_id()
2
```

\textbf{sage.groups.perm_gps.permgroup_named.PrimitiveGroups(d=None)}  
Return the set of all primitive groups of a given degree \(d\)

\textbf{INPUT:}
• \(d\) – an integer (optional)

\textbf{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.

\textbf{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
```

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!
```

\textbf{Todo:} This enumeration helper could be extended based on \texttt{PrimitiveGroupsIterator} in GAP. This method allows to enumerate groups with specified properties such as transitivity, solvability, \ldots, without creating all groups.
class sage.groups.perm_gps.permgroup_named.PrimitiveGroupsAll
Bases: sage.sets.disjoint_unionEnumeratedSets

The infinite set of all primitive groups up to isomorphisms.

EXAMPLES:

```python
sage: L = PrimitiveGroups(); L
Primitive Groups
sage: L.category()  
Category of facade infinite enumerated sets
sage: L.cardinality()  
+Infinity

sage: p = L.__iter__()
```

```python
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))
```

class sage.groups.perm_gps.permgroup_named.PrimitiveGroupsOfDegree(n)
parent

The set of all primitive groups of a given degree up to isomorphisms.

EXAMPLES:

```python
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)
```

We write the cardinality of all primitive groups of degree 5:

```python
sage: for G in PrimitiveGroups(5):
....:    print(G.cardinality())
5
10
20
60
120
```

cardinality()

Return the cardinality of self.

OUTPUT:

An integer. The number of primitive groups of a given degree up to isomorphism.

EXAMPLES:

```python
sage: PrimitiveGroups(0).cardinality()  
1
sage: PrimitiveGroups(2).cardinality()  
1
```
class sage.groups.perm_gps.permgroup_named.QuaternionGroup
Bases: sage.groups.perm_gps.permgroup_named.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 \{±1, ±I, ±J, ±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*I == neg_one and K*I == neg_one
True
sage: J*I == neg_one*K
True
sage: Q.center().order() == 2
True
sage: neg_one in Q.center()
True
```

AUTHOR:

• Rob Beezer (2009-10-09)

class sage.groups.perm_gps.permgroup_named.SemidihedralGroup(m)
Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique
The semidihedral group of order $2^m$.

INPUT:

• m - a positive integer; the power of 2 that is the group’s order
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 $2^{m-2}$.

```python
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
```

AUTHOR:
- Kevin Halasz (2012-8-7)

```python
class sage.groups.perm_gps.permgroup_named.SplitMetacyclicGroup(p, m)
Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique
The split metacyclic group of order $p^m$.

INPUT:
- p – a prime number that is the prime underlying this p-group
- m – a 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.

```python
sage: G = SplitMetacyclicGroup(2,8)
sage: G.order() == 2**8
True
sage: G.is_abelian()
False
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
```

AUTHOR:

- Kevin Halasz (2012-8-7)

class sage.groups.perm_gps.permgroup_named.SuzukiGroup(q, name='a')

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

The Suzuki group over GF(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 os not simple.)

- name -- (default: ‘a’) variable name of indeterminate of finite field GF(q)
A Suzuki group.

**Note:** This group is also available via `groups.permutation.Suzuki()`.

**EXAMPLES:**

```python
sage: SuzukiGroup(8)
sage: print(SuzukiGroup(8))
The Suzuki group over Finite Field in a of size 2^3
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
```

**REFERENCES:**

- Wikipedia article Group_of_Lie_type#Suzuki-Ree_groups
- `base_ring()`

**EXAMPLES:**

```python
sage: G = SuzukiGroup(32, name='alpha')
sage: G.base_ring()
Finite Field in alpha of size 2^5
```

**class** `sage.groups.perm_gps.permgroup_named.SuzukiSporadicGroup`

Bases: `sage.groups.perm_gps.permgroup_named.PermutationGroup_unique`

Suzuki Sporadic Group

**EXAMPLES:**

```python
sage: G = groups.permutation.SuzukiSporadic(); G
# optional - gap_packages internet
Sporadic Suzuki group acting on 1782 points
```

**class** `sage.groups.perm_gps.permgroup_named.SymmetricGroup` *(domain=None)*

Bases: `sage.groups.perm_gps.permgroup_named.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:**

- $n$ – a positive integer, or list or tuple thereof
Note: This group is also available via `groups.permutation.Symmetric()`.

**EXAMPLES:**

```python
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
```

**Element**

alias of `sage.groups.perm_gps.permgroup_element.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:**

```python
sage: S4 = SymmetricGroup(4)
sage: S4.algebra(QQ)
Symmetric group algebra of order 4 over Rational Field

sage: S3 = SymmetricGroup([1,2,3])
sage: A = S3.algebra(QQ); A
Symmetric group algebra of order 3 over Rational Field
sage: a = S3.an_element(); a
(2,3)
sage: A(a)
(2,3)
```
We illustrate the choice of the category:

```ruby
sage: A.category()
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())

sage: A.category()
Category of finite dimensional unital cellular semigroup algebras
    over Rational Field
```

In the following case, a usual group algebra is returned:

```ruby
sage: S = SymmetricGroup([2,3,5])
sage: a = S.an_element(); a
(3,5)
sage: S.algebra(QQ)(a)
(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:**

```ruby
sage: A = SymmetricGroup([2,3,7]); A.cartan_type()
['A', 2]
sage: A = SymmetricGroup([3,5,7]); 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:**

```ruby
sage: G = SymmetricGroup(5)
sage: g = G((1,2,3,4))
sage: G.conjugacy_class(g)
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:**

```ruby
sage: G = SymmetricGroup(5)
sage: G.conjugacy_classes()
[Conjugacy class of cycle type [1, 1, 1, 1, 1] in
    Symmetric group of order 5! as a permutation group,
```

(continues on next page)
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()
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 \( \lambda \) 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()
[(), (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()
[(), ('a','b'), ('a','b','c')]
```

coxeter_matrix()
Return the Coxeter matrix of self.

EXAMPLES:

```
sage: A = SymmetricGroup([2,3,7,'a']); A.coxeter_matrix()
[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])
sage: S.index_set()
(3, 1, 4)

**major_index** *(parameter=None)*

Return the major index generating polynomial of self, which 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:**

```python
sage: S4 = SymmetricGroup(4)
sage: S4.major_index()
qu^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)
t^6 + 3*t^5 + 5*t^4 + 6*t^3 + 5*t^2 + 3*t + 1
```

**reflections()**

Return the list of all reflections in self.

**EXAMPLES:**

```python
sage: A = SymmetricGroup(3)
sage: A.reflections()
[(1,2), (1,3), (2,3)]
```

**simple_reflection**(i)

For i in the index set of self, this returns the elementary transposition \( s_i = (i, i+1) \).

**EXAMPLES:**

```python
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)}
```

**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 [(7,8), (5,6), (3,4), (1,2)] 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

class sage.groups.perm_gps.permgroup_named.TransitiveGroup(d, n)
Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

The transitive group from the GAP tables of transitive groups.

INPUT:
- d – non-negative 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

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}
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
```

**degree()**

Return the degree of this permutation group

**EXAMPLES:**

```
sage: TransitiveGroup(8, 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
```

```
sage.groups.perm_gps.permgroup_named.TransitiveGroups(d=None)

INPUT:

- d – an integer (optional)

Returns 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
```

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
```

class sage.groups.perm_gps.permgroup_named.TransitiveGroupsAll

Bases: sage.sets.disjoint_union Enumerated Sets.DisjointUnionEnumeratedSets
The infinite set of all transitive groups up to isomorphisms.

**EXAMPLES:**

```python
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)
```

class `sage.groups.perm_gps.permgroup_named.TransitiveGroupsOfDegree(n)`

Bases: `sage.structure.unique_representation.CachedRepresentation`, `sage.structure.parent Parent`

The set of all transitive groups of a given (small) degree up to isomorphism.

**EXAMPLES:**

```python
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
```

We write the cardinality of all transitive groups of degree 5:

```python
sage: for G in TransitiveGroups(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:**

```python
sage: TransitiveGroups(0).cardinality()
1
sage: TransitiveGroups(2).cardinality()
```

(continues on next page)
24.5 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 trac ticket #26750)
• Sebastian Oehms (2019-02): Implemented gap() properly (trac ticket #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:

```python
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)
```
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
```

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
```

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()
432502003274489856000
sage: 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
```

```
class sage.groups.perm_gps.permgroup_element.PermutationGroupElement
    Bases: sage.structure.element.MultiplicativeGroupElement

    An element of a permutation group.

    EXAMPLES:

    sage: G = PermutationGroup('[(1,2,3)(4,5)]')
sage: G
    Permutation Group with generators [(1,2,3)(4,5)]
sage: g = G.random_element()
sage: g in G
```
True

```python
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
```

This example illustrates how permutations act on multivariate polynomials.

```python
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
```

`cycle_string(singletons=False)`

Return string representation of this permutation.

```python
sage: g = PermutationGroupElement([(1,2,3),(4,5)])
sage: g.cycle_string()
'(1,2,3)(4,5)'
```

`cycle_tuples(singletons=False)`

Return self as a list of disjoint cycles, represented as tuples rather than permutation group elements.

```python
sage: p = PermutationGroupElement('(2,6)(4,5,1)')
sage: p.cycle_tuples()
[(1, 4, 5), (2, 6)]
```

(continues on next page)
sage: 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)]
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')]

\textbf{cycle_type}(\texttt{singletons=True, as_list=False})

Return the partition that gives the cycle type of \( g \) as an element of \( \text{self} \).

\textbf{INPUT:}

\begin{itemize}
  \item \( g \) – an element of the permutation group \( \text{self.parent()} \)
  \item \texttt{singletons} – \texttt{True} or \texttt{False} depending on whether on or not trivial cycles should be counted (default: \texttt{True})
  \item \texttt{as_list} – \texttt{True} or \texttt{False} depending on whether the cycle type should be returned as a list or as a \texttt{Partition} (default: \texttt{False})
\end{itemize}

\textbf{OUTPUT:}

A \texttt{Partition}, or list if \texttt{is_list} is \texttt{True}, giving the cycle type of \( g \)

If speed is a concern then \texttt{as_list=\texttt{True}} should be used.

\textbf{EXAMPLES:}

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))').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]

cycles()

Return self as a list of disjoint cycles.

\textbf{EXAMPLES:}

sage: G = PermutationGroup(['(1,2,3)(4,5,6,7)'])
sage: g = G.0
sage: g.cycles()
[(1,2,3), (4,5,6,7)]
sage: a, b = g.cycles()
sage: a(1), b(1)
(2, 1)

**dict()**

Returns a dictionary associating each element of the domain with its image.

**EXAMPLES:**

```python
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}
```

**domain()**

Returns the domain of self.

**EXAMPLES:**

```python
sage: G = SymmetricGroup(4)
sage: x = G([2,1,4,3]); x
(1,2)(3,4)
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]
```

**gap()**

Returns self as a libgap element

**EXAMPLES:**

```python
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
```

(continues on next page)
has_descent(i, side='right', positive=False)

INPUT:

• i: an element of the index set
• side: “left” or “right” (default: “right”)
• positive: a boolean (default: False)

Returns 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).

EXAMPLES:

```python
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()
```

The symmetric group acting on a set not of the form (1, ..., n) is also supported:

```python
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()
```

inverse()

Returns 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:
There are several ways (syntactically) to get an inverse of a permutation group element.

```python
sage: s = PermutationGroupElement("(1,2,3)(4,5)"
(1,3,2)(4,5)
sage: A = AlternatingGroup(4)
sage: t = A("(1,2,3)"
(1,3,2)
```

```python
sage: s = PermutationGroupElement("(1,2,3,4)(6,7,8)"
(1,3,2)(4,5) == s^-1
True
sage: s = PermutationGroupElement("(1,2,3,4)(6,7,8)"
(1,3,2)(4,5) == ~s
True
```

### matrix()
Returns deg x deg permutation matrix associated to the permutation self

**EXAMPLES:**

```python
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]
```

### multiplicative_order()
Return the order of this group element, which is the smallest positive integer $n$ for which $g^n = 1$.

**EXAMPLES:**

```python
sage: s = PermutationGroupElement(''(1,2)(3,5,6)'
6
```

```python
order is just an alias for multiplicative_order:
```

```python
sage: s.order()
6
```

### orbit($n$, sorted=True)
Returns the orbit of the integer $n$ under this group element, as a sorted list.

**EXAMPLES:**

```python
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: g = G.gen(0)
sage: g.orbit(4)
[4, 5]
sage: g.orbit(3)
```
(continues on next page)


```
[1, 2, 3]
sage: g.orbit(10)
[10]
```

```
sage: s = SymmetricGroup(['a', 'b']).gen(0); s
('a', 'b')
sage: s.orbit('a')
['a', 'b']
```

**sign()**

Returns 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
```

**ALGORITHM:** Only even cycles contribute to the sign, thus

\[
sign(\sigma) = (-1)^{\sum_{\text{c}} \text{len}(\text{c}) - 1}
\]

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')
```

**word_problem**(words, display=True, as_list=False)

Try to solve the word problem for self.

**INPUT:**

- words – a 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 \(\text{words}\) of elements of \(G\). If \(g\) is in \(H\), this function returns an expression for \(g\) as a word in the elements of \(\text{words}\) 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:

```python
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)')
```

```python
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)')
```

```python
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]]
```

class sage.groups.perm_gps.permgroup_element.SymmetricGroupElement

Bases: sage.groups.perm_gps.permgroup_element.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:

```python
sage: S = SymmetricGroup(3)
sage: [x.absolute_length() for x in S]
[0, 2, 2, 1, 1, 1]
```

**has_left_descent(i)**

Return whether i is a left descent of self.

EXAMPLES:

```python
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]
```

sage.groups.perm_gps.permgroup_element.is_PermutationGroupElement(x)

Returns True if x is a PermutationGroupElement.

EXAMPLES:

```python
sage: p = PermutationGroupElement([[(1,2),(3,4,5)]])
sage: from sage.groups.perm_gps.permgroup_element import is_PermutationGroupElement
```

(continues on next page)
sage: is_PermutationGroupElement(p)
True

sage.groups.perm_gps.permgroup_element.make_permgroup_element(G, x)
Returns 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:

```python
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)
```

sage.groups.perm_gps.permgroup_element.make_permgroup_element_v2(G, x, domain)
Returns 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:

```python
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)
```

### 24.6 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:

```python
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

class sage.groups.perm_gps.permgroup_morphism.PermutationGroupMorphism
Bases: sage.categories.morphism.Morphism
A set-theoretic map between PermutationGroups.

image(J)
J must be a subgroup of G. Computes the subgroup of H which is the image of J.

EXAMPLES:

sage: G = CyclicPermutationGroup(4)
sage: H = DihedralGroup(4)
sage: g = G([(1,2,3,4)])
sage: phi = PermutationGroupMorphism(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)

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

Check that trac ticket #28324 is fixed:

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

kernel()
Returns the kernel of this homomorphism as a permutation group.

EXAMPLES:
```python
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)
```

```python
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
```

```
class sage.groups.perm_gps.permgroup_morphism.PermutationGroupMorphism_from_gap(G, H, gap_hom)

Bases: sage.groups.perm_gps.permgroup_morphism.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:

```python
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:
  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
```
```
```
```
```
```
```
```
```
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)

AUTHORS:
- David Joyner (2006-02)
sage.groups.perm_gps.permgroup_morphism.is_PermutationGroupMorphism(f)
Returns 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)
True

24.7 Rubik’s cube group functions

Note: “Rubik’s 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 $x - y$.

```python
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 |
+--------------+
```

**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
- Robert Miller (2007, 2008): editing, cleaned up display2d
- David Joyner (2007-09): rewrote docstring for CubeGroup’s “solve”.

**REFERENCES:**

```python
class sage.groups.perm_gps.cubegroup.CubeGroup
Bases: sage.groups.perm_gps.permgroup.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 $\text{SymmetricGroup}(48)$, 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 |
+--------------+

sage: rubik

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)

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)

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)

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)
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)
```

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)
```

display2d$(mv)$
Print the 2d representation of self.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.display2d("R")

+--------------+-------------+------------+-------------+-------------+------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-------------+-----------
Now the dictionary of the state obtained after making the move \( R \) followed by \( L \):

```python
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]])]
```

facets\( (g=None) \)

Return the set of facets on which the group acts. This function is a “constant”.

EXAMPLES:

```python
sage: rubik = CubeGroup()
sage: rubik.facets() == list(range(1,49))
True
```

gen_names()

Return the names of the generators.

EXAMPLES:

```python
sage: rubik = CubeGroup()
sage: rubik.gen_names()
['B', 'D', 'F', 'L', 'R', 'U']
```

legal\( (state, \text{mode}='\text{quiet}') \)

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:

```python
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
```

move\( (mv) \)

Return the group element and the reordered list of facets, as moved by the list \( mv \) (read left-to-right)

INPUT:

- \( mv \) – A string of the form \( Xa*Yb* \ldots \), where \( X, Y, \ldots \) are in \( R, L, F, B, U, D \) and \( a, b, \ldots \) are integers.

EXAMPLES:
sage: rubik = CubeGroup()
sage: rubik.move("")[0]
()
sage: rubik.move("R")[0]
sage: rubik.R()

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, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 6, 36, 4, 38, 39, 1, 40, 42, 43, 45, 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
{'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")
sage: C.parse("L^4")
(continues on next page)
plot3d_cube\(mv, title=\text{True}\)
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:
- \(mv\) – A string in the Singmaster notation
- \(\text{title}\) – (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 \(\text{show(P)}\) to view them.

EXAMPLES:

```sage
sage: rubik = CubeGroup()
```

plot_cube\(mv, title=\text{True}, colors=\{(1, 0.63, 1), (1, 1, 0), (0, 1, 0), (1, 0.6, 0.3), (0, 0, 1)\}\)
Input the move \(mv\), as a string in the Singmaster notation, and output the 2D plot of the cube in that state. Type \(\text{P.show()}\) to display any of the plots below.

EXAMPLES:

```sage
sage: rubik = CubeGroup()
sage: # (R^2U^2)^3 permutes 2 pairs of edges (uf,ub)(fr,br)
sage: P = rubik.plot_cube("R^2*D^2*R^3*D^2*F^2*D^3*F^2*D^3")
```

repr2d\(mv\)
Displays a 2D map of the Rubik’s cube after the move \(mv\) has been made. Nothing is returned.

EXAMPLES:

```sage
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 |
(continues on next page)"
You can see the right face has been rotated but not the left face.

You can see the right face has been rotated but not the left face.

```
sage: print(rubik.repr2d("R"))
```

```
| 1 2 38 |
| 4 top 36 |
| 6 7 33 |
```

```
| 9 10 11 |
```

```
| 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 \(4.3 \times 10^{19}\) 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)
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: sage.structure.sage_object.SageObject

The Rubik’s cube (in a given state).

EXAMPLES:

```python
sage: C = RubiksCube().move("R U R
''")
sage: C.show3d()
```

```python
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()
sage: C.solve(algorithm='gap') # long time
'L*R'
sage: 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` – (Default True) Boolean to display stickers

EXAMPLES:
sage: C = RubiksCube("R*U")

sage: C.cubie(0.15, 0.025, 0, 0, 0, C.colors*3)
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]

move(g)

Move the Rubik’s cube by g.

EXAMPLES:

sage: RubiksCube().move("R*U") == RubiksCube("R*U")
True

plot()

Return a plot of self.

EXAMPLES:

sage: C = RubiksCube("R*U")

sage: C.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()
Graphics3d Object

scramble(moves=30)

Scramble the Rubik’s cube.

EXAMPLES:

sage: C = RubiksCube()

sage: C.scramble()  # random
+-------------+
<p>| 38 29 35 |
| 20 top 42 |</p>
<table>
<thead>
<tr>
<th>11 44 30</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 13 17</td>
</tr>
<tr>
<td>4 left 18</td>
</tr>
<tr>
<td>33 31 40</td>
</tr>
</tbody>
</table>
(continues on next page)
show()
Show a plot of self.

EXAMPLES:

```python
sage: C = RubiksCube("R^U")
sage: C.show()
```

show3d()
Show a 3D plot of self.

EXAMPLES:

```python
sage: C = RubiksCube("R^U")
sage: C.show3d()
```

solve(algorithm='hybrid', timeout=15)
Solve the Rubik's cube.

INPUT:

- algorithm – must be one of the following:
  - hybrid - 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. Otherwise, the gap algorithm is used.

EXAMPLES:

```python
sage: C = RubiksCube("R U F L B D")
sage: C.solve()                     # optional - rubiks
'R U F L B D'
```

Dietz's program is much faster, but may give highly non-optimal solutions:

```python
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 L D' B L B L B' L' U 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 B' L F' B D' B' L D B' B' L D' L D B' B' U' U' L' B' D' F' F' L
D F'"
sage: C2 = RubiksCube(s)            # optional - rubiks
sage: C == C2                       # optional - rubiks
True
```

24.7. Rubik's cube group functions
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

sage.groups.perm_gps.cubegroup.color_of_square(facet, colors=["purple", "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'

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)
  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]

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)]

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())

(continues on next page)
sage.groups.perm_gps.cubegroup.index2singmaster(facet)
Translate index used (eg, 43) to Singmaster facet notation (eg, fdr).

**EXAMPLES:**

```python
sage: from sage.groups.perm_gps.cubegroup import index2singmaster
sage: index2singmaster(41)
'dlf'
```

sage.groups.perm_gps.cubegroup.inv_list(lst)
Input a list of ints 1, ..., m (in any order), outputs inverse perm.

**EXAMPLES:**

```python
sage: from sage.groups.perm_gps.cubegroup import inv_list
sage: L = [2,3,1]
sage: 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])

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 (i.e., addition of Graphics objects is not commutative).

The following example produced a green-colored square with vertices at the points indicated.

**EXAMPLES:**

```python
sage: from sage.groups.perm_gps.cubegroup import polygon_plot3d, green
tilsage: P = polygon_plot3d([[1,3,1], [2,3,1], [2,3,2], [1,3,2], [1,3,1]], rgbcolor=green)
```

sage: from sage.groups.perm_gps.cubegroup import rotation_list
sage: rotation_list(30, 45)
[0.49999999999999994, 0.7071067811865475, 0.8660254037844387, 0.7071067811865476]

sage: from sage.groups.perm_gps.cubegroup import xproj
sage: xproj(1,2,3, rot)
0.6123724356957945

sage: from sage.groups.perm_gps.cubegroup import yproj
sage: yproj(1,2,3, rot)
1.378497416975604
24.8 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: sage.groups.perm_gps.symgp_conjugacy_class.SymmetricGroupConjugacyClassMixin,
sage.groups.conjugacy_classes.ConjugacyClass

A conjugacy class of the permutations of \( n \).

INPUT:

• \( P \) – the permutations of \( n \)
• \( \text{part} \) – a partition or an element of \( P \)

\text{set}()

The set of all elements in the conjugacy class \( \text{self} \).

EXAMPLES:

```python
sage: G = Permutations(3)
sage: g = G([2, 1, 3])
sage: C = G.conjugacy_class(g)
sage: S = [[1, 3, 2], [2, 1, 3], [3, 2, 1]]
sage: C.set() == Set(G(x) for x in S)
True
```

class sage.groups.perm_gps.symgp_conjugacy_class.SymmetricGroupConjugacyClass(group, part)

Bases: sage.groups.perm_gps.symgp_conjugacy_class.SymmetricGroupConjugacyClassMixin,
sage.groups.conjugacy_classes.ConjugacyClassGAP

A conjugacy class of the symmetric group.

INPUT:

• \( \text{group} \) – the symmetric group
• \( \text{part} \) – a partition or an element of \( \text{group} \)

\text{set}()

The set of all elements in the conjugacy class \( \text{self} \).

EXAMPLES:

```python
sage: G = SymmetricGroup(3)
sage: g = G((1,2))
sage: C = G.conjugacy_class(g)
sage: S = [(2,3), (1,2), (1,3)]
sage: C.set() == Set(G(x) for x in S)
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.

\text{partition}()

Return the partition of \( \text{self} \).
EXAMPLES:

```python
sage: G = SymmetricGroup(5)
sage: g = G([(1,2), (3,4,5)])
sage: C = G.conjugacy_class(g)
```

`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` – (optional, 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:

```python
sage: from sage.groups.perm_gps.symgp_conjugacy_class import conjugacy_class_iterator
sage: for p in conjugacy_class_iterator([2,2]): print(p)
[(1, 2), (3, 4)]
[(1, 4), (2, 3)]
[(1, 3), (2, 4)]
```

In order to get permutations, one just has to wrap:

```python
sage: S = SymmetricGroup(5)
sage: for p in conjugacy_class_iterator([3,2]): print(S(p))
(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)
```

Check that the number of elements is the number of elements in the conjugacy class:

```python
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))
True
```

It is also possible to specify any underlying set:

```python
sage: it = conjugacy_class_iterator([2,2,2], 'abcdef')
sage: sorted(flatten(next(it)))
['a', 'b', 'c', 'd', 'e', 'f']
sage: all(len(x) == 2 for x in next(it))
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`.  

346 Chapter 24. Permutation Groups
Let $\lambda$ 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 $\ell$ is the length (or number of parts) of $\lambda$.

**INPUT:**

- `part` – partition
- `G` – a symmetric group

**EXAMPLES:**

```python
sage: from sage.groups.perm_gps.symgp_conjugacy_class import default_representative
sage: S = SymmetricGroup(4)
sage: for p in Partitions(4):
    ....:     print(default_representative(p, S))
(1,2,3,4)
(1,2,3)
(1,2)(3,4)
(1,2)
()```

24.8. Conjugacy Classes Of The Symmetric Group 347
CHAPTER TWENTYFIVE

MATRIX AND AFFINE GROUPS

25.1 Library of Interesting Groups

Type $\texttt{groups.matrix.<tab>}$ to access examples of groups implemented as permutation groups.

25.2 Base classes for Matrix Groups

Loading, saving, ... works:

```python
sage: G = GL(2, 5); G
General Linear Group of degree 2 over Finite Field of size 5
sage: TestSuite(G).run()

sage: g = G.1; g
[4 1]
[4 0]
sage: TestSuite(g).run()
```

We test that trac ticket #9437 is fixed:

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

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).
- Sebastian Oehms (2018-07): Add subgroup() and ambient() see trac ticket #25894

```
class sage.groups.matrix_gps.matrix_group.MatrixGroup_base
    Bases: sage.groups.group.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])
sage: 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))
sage: G.as_matrix_group()
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 3*a ], [ 0 3*a + 1 0 0 ] )
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 ] )
```

**sign_representation**

Return the sign representation of self over base_ring.

**WARNING:** assumes self is a matrix group over a field which has embedding over real numbers.

**INPUT:**

- base_ring – (optional) the base ring; the default is Z
- side – ignored

**EXAMPLES:**

```
sage: G = GL(2, QQ)
sage: V = G.sign_representation()
```

(continues on next page)
```
sage: e = G.an_element()
sage: e
[1 0]
[0 1]
sage: V._default_sign(e)
1
sage: m2 = V.an_element()
sage: m2
2*B['v']
sage: m2*e
2*B['v']
sage: m2*e*e
2*B['v']
```

**subgroup(generators, check=True)**

Return the subgroup generated by the given generators.

INPUT:

- **generators** – a list/tuple/iterable of group elements of self
- **check** – boolean (optional, default: True). Whether to check that each matrix is invertible.

OUTPUT: The subgroup generated by `generators` as an instance of `FinitelyGeneratedMatrixGroup_gap`.

EXAMPLES:

```python
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
```

```python
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
```

**class** `sage.groups.matrix_gps.matrix_group.MatrixGroup_gap`

```
Bases: sage.groups.libgap_mixin.GroupMixinLibGAP, sage.groups.matrix_gps.matrix_group.
MatrixGroup_generic, sage.groups.libgap_wrapper.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 *sage.groups.matrix_gps.group_element.MatrixGroupElement_gap*

**structure_description**(*G*, *latex*=*False*)

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

This method 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** – a 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:

```python
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}
```

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

```python
sage: D3 = DihedralGroup(3)
sage: D3.structure_description()
'S3'
```

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

```python
sage: D4 = DihedralGroup(4)
sage: D4.structure_description()
'D4'
```

Works for finitely presented groups (trac ticket #17573):
sage: F.<x, y> = FreeGroup()
sage: G=F / [x^2*y^−1, x^3*y^2, x*y*x^-1*y^-1]
sage: G.structure_description()
'\text{C}_7'

And matrix groups (trac ticket #17573):

```python
sage: groups.matrix.GL(4,2).structure_description()
'A_8'
```

class sage.groups.matrix_gps.matrix_group.MatrixGroup_generic(degree, base_ring, category=None)

Bases: sage.groups.matrix_gps.matrix_group.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 sage.groups.matrix_gps.group_element.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:

```python
sage: SU(5,5).degree()
5
```

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:

```python
sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: G = MatrixGroup([[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
```

sage.groups.matrix_gps.matrix_group.is_MatrixGroup(x)

Test whether x is a matrix group.

EXAMPLES:

```python
sage: from sage.groups.matrix_gps.matrix_group import is_MatrixGroup
sage: is_MatrixGroup(MatrixSpace(QQ,3))
False
```
is_MatrixGroup(Mat(QQ,3))
False
is_MatrixGroup(GL(2,ZZ))
True
is_MatrixGroup(MatrixGroup([matrix(2,[1,1,0,1]))
True

## 25.3 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] \\
[0 1], [1 1] \\
[0 1], [0 1]
)  
```

```sage
g = G([[1,1],[0,1]])  
h = G([[1,2],[0,1]])  
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] \\
[0 1], [1 1] \\
[0 1], [0 1]
)'
and
'Matrix group over Finite Field of size 3 with 2 generators
\([1 0] \\
[0 1], [1 1] \\
[0 1], [0 1]
)'
```

```sage
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...
```

(continues on next page)
AUTHORS:

- David Joyner (2006-05): initial version
- David Joyner (2006-05): various modifications to address William Stein’s TODO’s.
- Volker Braun (2013-1): port to new Parent, libGAP.
- Travis Scrimshaw (2016-01): reworks class hierarchy in order to cythonize

```
[1 0] [1 1]
[0 1], [0 1]
```

AUTHORS:

- David Joyner (2006-05): initial version
- David Joyner (2006-05): various modifications to address William Stein’s TODO’s.
- 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_gap`

Bases: `sage.groups.libgap_wrapper.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` – bool (default: True); if True does some type checking
- `convert` – bool (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]]
```

**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()
```

(continues on next page)
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
```

Matrices have extra functionality that matrix group elements do not have:

```
sage: 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)
```

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]
```
```python
sage: 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` – a 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:**

```python
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

Next we construct a more complicated element of the group from the generators:

```python
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]
]
```

(continues on next page)
We solve the word problem using some different generators:

\[
\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}^{-1} \times \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}^{-1} \times \begin{bmatrix} 0 & 4 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}^{-1}
\]

We try some elements that don’t actually generate the group:

\[
\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}^{-1} * \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}^{-1} * \begin{bmatrix} 0 & 4 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}^{-1}
\]

AUTHORS:

• David Joyner and William Stein
• David Loeffler (2010): fixed some bugs
• Volker Braun (2013): LibGAP

class sage.groups.matrix_gps.group_element.MatrixGroupElement_generic

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 – bool (default: True); if True, then does some type checking
• convert – bool (default: True); if True, then convert \( M \) to the right matrix space

EXAMPLES:
inverse()
Return the inverse group element

OUTPUT:
A matrix group element.

EXAMPLES:

```
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
```

is_one()
Return whether self is the identity of the group.

EXAMPLES:

```
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
```

list()
Return list representation of this matrix.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], base_ring=ZZ)
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]]]```
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: W = CoxeterGroup(['A',3], base_ring=IntegerRing)
sage: g = W.an_element()
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
```

Matrices have extra functionality that matrix group elements do not have:

```
sage: g.matrix().charpoly('t')
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('hellooo')
False
sage: G = GL(2,3)
sage: is_MatrixGroupElement(G.an_element())
True
```

## 25.4 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()
([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
```

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).
- Volker Braun (2013-1) port to new Parent, libGAP.
- Sebastian Oehms (2018-07): Added _permutation_group_element_ (Trac #25706)
- Sebastian Oehms (2019-01): Revision of trac ticket #25706 (trac ticket #26903 and trac ticket #27143).

```python
class sage.groups.matrix_gps.finitely_generated.FinitelyGeneratedMatrixGroup_gap(
    degree, base_ring, libgap_group, ambient=None, category=None)
```

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] [3 4], [1  3] [10  0] )
sage: type(G)
<class 'sage.groups.matrix_gps.finitely_generated.FinitelyGeneratedMatrixGroup_gap'
˓→with_category'>
sage: 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:

```python
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
```

A finite subgroup of GL(12,Z) as a permutation group:

```python
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 == Psmaller  # see the note below
True
sage: Psmaller = G.as_permutation_group(algorithm="smaller")
sage: P == Psmaller
False
sage: P.cardinality()
21499084800
sage: P.degree()
144
sage: Psmaller.cardinality()
21499084800
sage: Psmaller.degree()
80
```

**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 GF(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
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()
[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]]; gen2 = [[1,0],[0,-1]]
\rightarrow
sage: gen3 = [[-1,0], [0,1]]
sage: G = MatrixGroup([MS(gen1),MS(gen2),MS(gen3)])
sage: G.cardinality()
12
sage: G.invariant_generators()
[x1^2 + 3*x2^2, x1^6 + 15*x1^4*x2^2 + 15*x1^2*x2^4 + 33*x2^6]
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()
[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:**
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:

- \( \text{degree} \) – a positive integer
- \( \text{chi} \) – (default: trivial character) a linear group character of this group
- \( R \) – (optional) a polynomial ring

OUTPUT: list of polynomials

EXAMPLES:

```
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]
```

```
sage: R.<x,y,z> = QQ[]
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
```

```
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)
[]
```

```
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
```
```python
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*i^3 + 3*i^2 - 8*i - 4)*x*z + (2*i^3 + 3*i^2 + 8*i + 3)*y*z,
 x^2 + (2*i^3 + 3*i^2 + 8*i + 3)*y^2 + (-2*i^3 + 3*i^2 - 8*i - 4)*z^2]
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]
```

**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:

```python
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)]
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)]
```

Type G.module_composition_factors(algorithm='verbose') to get a more verbose version.

For more on MeatAxe notation, see https://www.gap-system.org/Manuals/doc/ref/chap69.html

**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 $\lambda$ as a primitive $N$-th root of unity over $k$ and $\omega$ as a primitive $N$-th root of unity over $\mathbb{Q}$. 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
- `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()
Traceback (most recent call last):
  ... NotImplementedError: characteristic cannot divide group order
```

Tetrahedral Group:

```
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: 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
```
Octahedral Group:

```sage
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], [(1+i)/a,0, 0, (1-i)/a])
sage: Octa.molien_series(prec=30)
1 + t^8 + t^12 + t^16 + t^20 + 2*t^24 + t^26 + t^28 + O(t^30)
```

Icosahedral Group:

```sage
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)
```

```
sage: G = MatrixGroup(CyclicPermutationGroup(3))
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)
```

Sage: G = MatrixGroup(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)
```

```
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)
```

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 $\chi$ a linear character for $G$. Let $b$ be the ring of invariants of $G$ relative to $\chi$. Then the Reynold’s operator is a map $R$ from $K[x]$ into $b$ defined by

**INPUT:**

- poly – a polynomial
• chi – (default: trivial character) a linear group character of this group

OUTPUT: an invariant polynomial relative to $\chi$

AUTHORS:
Rebecca Lauren Miller and Ben Hutz

EXAMPLES:

```python
sage: S3 = MatrixGroup(SymmetricGroup(3))
sage: R.<x,y,z> = QQ[]
sage: f = x*y*z^3
sage: S3.reynolds_operator(f)
1/3*x^3*y*z + 1/3*x*y^3*z + 1/3*x*y*z^3

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

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

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
```
```python
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
```

Characteristic p>0 examples:

```python
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
```

```python
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: R.<w,x> = GF(7)[]
sage: f = w^5*x + x^6
sage: G.reynolds_operator(f, chi)
Traceback (most recent call last):
  ...  
NotImplementedError: nontrivial characters not implemented for characteristic >, ...
```

```python
sage: K = GF(3^2,'t')
sage: G = MatrixGroup([matrix(K,2,2, [0,K.gen(),1,0])])
sage: R.<x,y> = K[]
sage: f = -K.gen()*x
sage: G.reynolds_operator(f)
t*x + t*y
```

```python
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^8*x + t^8*y
```

25.4. Finitely Generated Matrix Groups
class sage.groups.matrix_gps.finitely_generated.FinitelyGeneratedMatrixGroup_generic(degree, base_ring, generator_matrices, category=None)

Bases: sage.groups.matrix_gps.matrix_group.MatrixGroup_generic

gen(i)
Return the i-th generator

OUTPUT:
The i-th generator of the group.

EXAMPLES:

```
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
```

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()
```

ngens()
Return the number of generators

370 Chapter 25. Matrix and Affine Groups
OUTPUT:

An integer. The number of generators.

EXAMPLES:

```
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
```

```
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 (optional, 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]
)
```

In the second example, the generators are a matrix over \( \mathbb{Z} \), 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]
[4 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 (  
```
Some groups are not supported, or do not have much functionality implemented:

```python
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'
```

```python
sage.groups.matrix_gps.finitely_generated.QuaternionMatrixGroupGF3()
The quaternion group as a set of 2 x 2 matrices over GF(3).
```

**Output:**
A matrix group consisting of 2 x 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 $J$.

```python
sage: from sage.groups.matrix_gps.finitely_generated import QuaternionMatrixGroupGF3
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]
```

```python
sage.groups.matrix_gps.finitely_generated.normalize_square_matrices(matrices)
Find a common space for all matrices.
```

**Output:**
A list of matrices, all elements of the same matrix space.
EXAMPLES:

```
from sage.groups.matrix_gps.finitely_generated import normalize_square_matrices
m1 = [[1, 2], [3, 4]]
m2 = [2, 3, 4, 5]
m3 = matrix(QQ, [[1/2, 1/3], [1/4, 1/5]])
m4 = MatrixGroup(m3).gen(0)
```

```
[ 1  2]
[3 4]
```

```
25.5 Homomorphisms Between Matrix Groups

Deprecated May, 2018; use `sage.groups.libgap_morphism` instead.

`sage.groups.matrix_gps.morphism.to_libgap(x)`
Helper to convert `x` to a LibGAP matrix or matrix group element.

Deprecated; use the `x.gap()` method or `libgap(x)` instead.

```
from sage.groups.matrix_gps.morphism import to_libgap
```
```
sage: to_libgap(GL(2,3).gen(0))
```
```
doctest:...: DeprecationWarning: this function is deprecated.
Use x.gap() or libgap(x) instead.
```
```
See https://trac.sagemath.org/25444 for details.
```
```
[ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ]
```
```
sage: to_libgap(matrix(QQ, [[1,2],[3,4]]))
```
```
[ [ 1, 2 ], [ 3, 4 ] ]
```

25.6 Matrix Group Homsets

AUTHORS:

- Volker Braun (2013-1) port to new Parent, libGAP

`sage.groups.matrix_gps.homset.is_MatrixGroupHomset(x)`
Test whether `x` is a matrix group homset.

```
from sage.groups.matrix_gps.homset import is_MatrixGroupHomset
```
```
sage: is_MatrixGroupHomset(4)
```
```
doctest:...: DeprecationWarning: Importing MatrixGroupHomset from here is deprecated.
If you need to use it, please import it directly from sage.groups.libgap_morphism
```

See https://trac.sagemath.org/25444 for details.
False

```python
sage: F = GF(5)
sage: gens = [matrix(F,2,[1,2, -1, 1]), matrix(F,2, [1,1, 0,1])]
sage: G = MatrixGroup(gens)
sage: from sage.groups.matrix_gps.homset import MatrixGroupHomset
sage: M = MatrixGroupHomset(G, G)
sage: is_MatrixGroupHomset(M)
True
```

### 25.7 Binary Dihedral Groups

**AUTHORS:**
- Travis Scrimshaw (2016-02): initial version

**class** \texttt{sage.groups.matrix_gps.binary_dihedral.BinaryDihedralGroup}(n)

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.groups.matrix_gps.finitely_generated.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

$$
\begin{align*}
g_1 &= \begin{pmatrix}
\zeta_{2n} & 0 \\
0 & \zeta_{2n}^{-1}
\end{pmatrix}, &
g_2 &= \begin{pmatrix}
0 & \zeta_{4} \\
\zeta_{4} & 0
\end{pmatrix},
\end{align*}
$$

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 $g_2$, $g_2g_1^{-1}$ and $g_1$ in the matrix group avatar.)

**REFERENCES:**
- [Dol2009]
- [Sun2010]
- Wikipedia article Dicyclic_group#Binary_dihedral_group

**cardinality()**

Return the order of \texttt{self}, which is $4n$.

**EXAMPLES:**

```python
sage: G = groups.matrix.BinaryDihedral(3)
sage: G.order()
12
```

**order()**

Return the order of \texttt{self}, which is $4n$.

**EXAMPLES:**

```python
sage: G = groups.matrix.BinaryDihedral(3)
sage: G.order()
12
```

## 25.8 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:**

- `sage.structure.unique_representation.UniqueRepresentation`
- `sage.groups.matrix_gps.finitely_generated.FinitelyGeneratedMatrixGroup_generic`

A Coxeter group represented as a matrix group.

Let \((W, S)\) be a Coxeter system. We construct a vector space \(V\) over \(\mathbb{R}\) 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 \(B(\alpha_s, \alpha_s) = -1\) if \(m_{st} = \infty\). Next we define a representation \(\sigma_s : V \rightarrow V\) 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 \(\sigma_s\) 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 \(\mathbb{Q}\) or a quadratic number field when possible.

For more on creating Coxeter groups, see `CoxeterGroup()`.

**Todo:** Currently the label \(\infty\) is implemented as \(-1\) in the Coxeter matrix.

**EXAMPLES:**

We can create Coxeter groups from Coxeter matrices:

```python
sage: W = CoxeterGroup([[1, 6, 3], [6, 1, 10], [3, 10, 1]])
sage: W
Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[ 1  6  3]
[ 6  1 10]
[ 3 10  1]
sage: W.gens()
```

(continues on next page)
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]

(continues on next page)
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 \texttt{None}) are treated as 3:

```
sage: G = Graph([(0,3,\texttt{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
```

Because there currently is no class for \(\mathbb{Z} \cup \{\infty\}\), labels of \(\infty\) are given by \(-1\) in the Coxeter matrix:

```
sage: G = Graph([(0,1,\texttt{None}), (1,2,4), (0,2,\texttt{oo})])
sage: W = CoxeterGroup(G)
sage: W.coxeter_matrix()
[ 1 3 -1]
[ 3 1 4]
[-1 4 1]
```

We can also create Coxeter groups from Cartan types using the \texttt{implementation} keyword:

```
sage: W = CoxeterGroup(['D',5], \texttt{implementation=\"reflection\")}
sage: 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], \texttt{implementation=\"reflection\")}
sage: W
Finite Coxeter group over Number Field in a with defining polynomial \(x^2 - 5\) with a_\infty= 2.236067977499790? with Coxeter matrix:
[1 3 2]
[3 1 5]
[2 5 1]
```

\textbf{class} \texttt{Element}

\textbf{Bases:} \texttt{sage.groups.matrix_gps.group_element.MatrixGroupElement}\_\texttt{generic}

A Coxeter group element.

\texttt{action_on_root_indices}(i, \texttt{side=\textquoteleft left\textquoteright})

Return the action on the set of roots.
The roots are ordered as in the output of the method \texttt{roots}.

EXAMPLES:

\begin{Verbatim}
sage: W = CoxeterGroup(['A',3], implementation="reflection")
sage: w = W.w0
sage: w.action_on_root_indices(0)
11
\end{Verbatim}

\texttt{canonical_matrix}()

Return the matrix of \texttt{self} in the canonical faithful representation, which is \texttt{self} as a matrix.

EXAMPLES:

\begin{Verbatim}
sage: W = CoxeterGroup(['A',3], implementation="reflection")
sage: a,b,c = W.gens()
sage: elt = a*b*c
sage: elt.canonical_matrix()
\begin{bmatrix}
0 & 0 & -1 \\
1 & 0 & -1 \\
0 & 1 & -1
\end{bmatrix}
\end{Verbatim}

\texttt{descents} \texttt{(} \texttt{side=} \texttt{\textquotesingle} right\textquotesingle{}, \texttt{index_set=} \texttt{None}, \texttt{positive=} \texttt{False} \texttt{)}

Return the descents of \texttt{self}, as a list of elements of the \texttt{index_set}.

INPUT:

- \texttt{index_set} – (default: all of them) a subset (as a list or iterable) of the nodes of the Dynkin diagram
- \texttt{side} – (default: 'right') 'left' or 'right'
- \texttt{positive} – (default: False) boolean

EXAMPLES:

\begin{Verbatim}
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]
\end{Verbatim}

\texttt{first_descent} \texttt{(} \texttt{side=} \texttt{\textquotesingle} right\textquotesingle{}, \texttt{index_set=} \texttt{None}, \texttt{positive=} \texttt{False} \texttt{)}

Return the first left (resp. right) descent of \texttt{self}, as an element of \texttt{index_set}, or \texttt{None} if there is none.

See \texttt{descents()} for a description of the options.

EXAMPLES:

\begin{Verbatim}
sage: W = CoxeterGroup(['A',3], implementation="reflection")
sage: a,b,c = W.gens()
sage: elt = b*a*c
sage: elt.first_descent()
\end{Verbatim}
sage: elt.first_descent(side='left')

\textbf{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 non-positive.

\textbf{INPUT:}
\begin{itemize}
  \item \(i\) – an element in the index set
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
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]
\end{verbatim}

\textbf{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.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: W = CoxeterGroup(['D',4])
sage: W.bilinear_form()  
[ [ 1 -1/2  0  0]
 [ -1/2  1 -1/2 -1/2]
 [  0 -1/2  1  0]
 [  0 -1/2  0  1]
\end{verbatim}

\textbf{canonical_representation()}

Return the canonical faithful representation of \(self\), which is \(self\).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: W = CoxeterGroup([[1,3],[3,1]])
sage: W.canonical_representation() is W
True
\end{verbatim}

\textbf{coxeter_matrix()}

Return the Coxeter matrix of \(self\).

\textbf{EXAMPLES:}
\begin{itemize}
\item \texttt{sage: W = CoxeterGroup([[1,3],[3,1]])}
\item \texttt{sage: W.coxeter_matrix()}
\begin{verbatim}
[1 3]
[3 1]
\end{verbatim}
\item \texttt{sage: W = CoxeterGroup(['H',3])}
\item \texttt{sage: W.coxeter_matrix()}
\begin{verbatim}
[1 3 2]
[3 1 5]
[2 5 1]
\end{verbatim}
\end{itemize}

\textbf{fundamental_weight(i)}

Return the fundamental weight with index \(i\).

\textbf{See also:}

\textit{fundamental_weights()}.

\textbf{EXAMPLES:}

\begin{itemize}
\item \texttt{sage: W = CoxeterGroup(['A',3], implementation='reflection')}
\item \texttt{sage: W.fundamental_weight(1)}
\begin{verbatim}
(3/2, 1, 1/2)
\end{verbatim}
\end{itemize}

\textbf{fundamental_weights()}

Return the fundamental weights for \texttt{self}.

This is the dual basis to the basis of simple roots.

The base ring must be a field.

\textbf{See also:}

\textit{fundamental_weight()}.

\textbf{EXAMPLES:}

\begin{itemize}
\item \texttt{sage: W = CoxeterGroup(['A',3], implementation='reflection')}
\item \texttt{sage: W.fundamental_weights()}
\begin{verbatim}
Finite family {1: (3/2, 1, 1/2), 2: (1, 2, 1), 3: (1/2, 1, 3/2)}
\end{verbatim}
\end{itemize}

\textbf{is_commutative()}

Return whether \texttt{self} is commutative.

\textbf{EXAMPLES:}

\begin{itemize}
\item \texttt{sage: CoxeterGroup(['A', 2]).is_commutative()}
\begin{verbatim}
False
\end{verbatim}
\item \texttt{sage: W = CoxeterGroup(['I',2])}
\item \texttt{sage: W.is_commutative()}
\begin{verbatim}
True
\end{verbatim}
\end{itemize}

\textbf{is_finite()}

Return True if this group is finite.

\textbf{EXAMPLES:}

\begin{itemize}
\item \texttt{[l for l in range(2, 9) if ....: CoxeterGroup([[1,3,2],[3,1,1],[2,1,1]]).is_finite()]
\end{itemize}
order()

Return the order of self.

If the Coxeter group is finite, this uses an iterator.

EXAMPLES:

\begin{verbatim}
sage: W = CoxeterGroup([[1,3],[3,1]])
sage: W.order()
6
sage: W = CoxeterGroup([[1,-1],[-1,1]])
sage: W.order()
+Infinity
\end{verbatim}

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:

\begin{verbatim}
reflections()
\end{verbatim}

EXAMPLES:

\begin{verbatim}
sage: W = CoxeterGroup(['A',3], implementation='reflection')
sage: W.positive_roots()
((1, 0), (1, 1, 0), (0, 1, 0), (1, 1, 1), (0, 1, 1), (0, 0, 1))
sage: W = CoxeterGroup(['I',5], implementation='reflection')
sage: W.positive_roots()
((1, 0), (-E(5)^2 - E(5)^3, 1),
(continues on next page)
reflections()

Return the set of reflections.

The order is the one given by positive_roots().

EXAMPLES:

```python
sage: W = CoxeterGroup(['A',2], implementation='reflection')
sage: list(W.reflections())
[[-1  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:

```python
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))
sage: W = CoxeterGroup(['I',5], implementation='reflection')
sage: len(W.roots())
10
```

simple_reflection(i)

Return the simple reflection \(s_i\).

INPUT:

- \(i\) – an element from the index set

EXAMPLES:
```python
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]
```

The simple roots are indexed by the simple root index method.

```python
sage: simple_root_index(i)
Return the index of the simple root \( \alpha_i \).
This is the position of \( \alpha_i \) in the list of all roots as given be \( \text{roots()} \).
```

**EXAMPLES:**

```python
sage: W = CoxeterGroup(['A',3], implementation='reflection')
sage: [W.simple_root_index(i) for i in W.index_set()]
[0, 2, 5]
```

---

## 25.9 Linear Groups

**EXAMPLES:**

```python
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'))
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: 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()
```

(continues on next page)
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` – a 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:

```python
sage: G = GL(6,GF(5))
sage: G.order()
11064475420000000000000000
sage: G.base_ring()
Finite Field of size 5
sage: G.category()
Category of finite groups
sage: TestSuite(G).run()

sage: G = GL(6, QQ)
sage: G.category()
Category of infinite groups
sage: TestSuite(G).run()
```

Here is the Cayley graph of (relatively small) finite General Linear Group:

```python
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, vertex_labels=False)  # long
˓→time
```
Graphics object consisting of 144 graphics primitives

```
sage: d.plot3d(color_by_label=True)  # long time
```
Graphics3d Object

```
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
sage: H.gens()
```

```
\[
\begin{pmatrix}
2 & 0 \\
0 & 1 \\
\end{pmatrix},
\begin{pmatrix}
2 & 1 \\
2 & 0 \\
\end{pmatrix}
\]
```

```
class sage.groups.matrix_gps.linear.LinearMatrixGroup_gap(degree, base_ring, special, sage_name, latex_string, gap_command_string, category=None)

Bases: sage.groups.matrix_gps.named_group.NamedMatrixGroup_gap, sage.groups.matrix_gps.linear.LinearMatrixGroup_generic

FinitelyGeneratedMatrixGroup_gap

The general or special linear group in GAP.
```
class sage.groups.matrix_gps.linear.LinearMatrixGroup_generic(degree, base_ring, special, sage_name, latex_string, category=None)

invariant_form=None)

Bases: sage.groups.matrix_gps.named_group.NamedMatrixGroup_generic

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:

• \( n \) – a positive integer.

• \( R \) – ring or an integer. If an integer is specified, the corresponding finite field is used.

• \( \text{var} \) – variable used to represent generator of the finite field, if needed.

EXAMPLES:

```sage
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: G.order()
195671259569146962015219062429586341124018007182049478916067369638713066737862363393519966343657677430907011270206265834819092046250232049187967718149558134226774650845658791865745408000000
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])
)
```

Next we compute generators for \( \text{SL}_3(\mathbb{Z}) \)

```sage
sage: G = SL(3, ZZ); G
Special Linear Group of degree 3 over Integer Ring
sage: G.gens()
([0 1 0] [0 0 1] [-1 0 0], [0 0 1], [0 0 1]
)

sage: TestSuite(G).run()
```

## 25.10 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 \( GO(n, R) \) 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 \( \text{e} \) 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:**

```python
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: 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 \_OG\_, option for user defined invariant bilinear form, and bug-fix in cmd-string for calling GAP (see trac ticket #26028)

\[\text{sage.groups.matrixgps.orthogonal.} \quad \text{GO}(n, R, e=0, \text{var='a'}, \text{invariant_form=None})\]

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 \texttt{groups.matrix.GO()}.

**INPUT:**
- \texttt{n} – integer; the degree
- \texttt{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
- \texttt{var} – (optional, default: ‘\(a\)’) variable used to represent generator of the finite field, if needed

---

**25.10. Orthogonal Linear Groups**
• invariant_form – (optional) instances being accepted by the matrix-constructor which define a $n \times n$
square matrix over $\mathbb{R}$ 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: 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]
)
```

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]
[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
```

(continues on next page)
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]]

class sage.groups.matrix_gps.orthogonal.OrthogonalMatrixGroup_gap(degree, base_ring, special, name, latex_string, gap_command_string, category=None)


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 0 0 0]
[0 0 0 0]
[0 0 2 0]
[0 0 0 2]
sage: G = GO(4, GF(7), +1)
sage: G.invariant_bilinear_form()
[0 0 0 0]
[0 0 0 0]
[0 0 2 0]
[0 0 0 2]
sage: G = SO(4, GF(7), -1)
sage: G.invariant_bilinear_form()
[0 0 0 0]
[0 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]
```

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]
```

(continues on next page)
sage: G = SO(4, GF(7), -1)
sage: G.invariant_quadratic_form()
\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

class sage.groups.matrix_gps.orthogonal.OrthogonalMatrixGroup_generic(  
degree, base_ring,  
special, sage_name,  
latex_string,  
category=None,  
invariant_form=None)

Bases: sage.groups.matrix_gps.named_group.NamedMatrixGroup_generic

General Orthogonal Group over arbitrary rings.

EXAMPLES:

```sage
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
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
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}\rightarrow\text{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
sage: GO(2,3,+1).invariant_bilinear_form()
\[
\begin{bmatrix}
0 & 1 \\
1 & 0
\end{bmatrix}
\]
sage: GO(2,3,-1).invariant_bilinear_form()
\[
\begin{bmatrix}
2 & 1 \\
1 & 1
\end{bmatrix}
\]
sage: G = GO(4, QQ)
```
**invariant_bilinear_form()**

Return the symmetric bilinear form preserved by `self`.

**OUTPUT:**

A matrix.

**EXAMPLES:**

```
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]
```

**invariant_quadratic_form()**

Return the symmetric bilinear form preserved by `self`.

**OUTPUT:**

A matrix.

**EXAMPLES:**

```
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]
```
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
- $\text{var}$ – (optional, default: 'a') variable used to represent generator of the finite field, if needed
- $\text{invariant\_form}$ – (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:

```python
sage: G = SO(3,GF(5))
sage: G
Special Orthogonal Group of degree 3 over Finite Field of size 5
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 = SO(3,GF(5))
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]
```

(continues on next page)
Using the `invariant_form` option:

```python
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
```

`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.
25.11 Groups of isometries.

Let \( M = \mathbb{Z}^n \) or \( \mathbb{Q}^n \), \( b : M \times M \to \mathbb{Q} \) 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")
sage: O = L.orthogonal_group()
sage: O
Group of isometries with 5 generators ( [-1 0 0 0] [0 0 0 1] [-1 -1 -1 -1] [1 1 0 0] [1 0 0 0] [0 -1 0 0] [0 1 0 0] [0 0 1 0] [0 0 1 0] [-1 -1 -1 -1] [0 0 -1 0] [0 0 1 0] [0 1 0 1] [0 1 0 1] [0 0 1 0] [0 0 0 -1], [1 0 0 0], [0 -1 -1 0], [0 -1 -1 0], [0 0 0 1] )
```

Basic functionality is provided by GAP:

```
sage: O.cardinality()
1152
sage: len(O.conjugacy_classes_representatives())
25
```

**AUTHORS:**

- Simon Brandhorst (2018-02): First created

```
class sage.groups.matrix_gps.isometries.GroupActionOnQuotientModule(MatrixGroup, quotient_module, is_left=False)
```

Bases: `sage.categories.action.Action`

Matrix group action on a quotient module from the right.

**INPUT:**

- `MatrixGroup` – the group acting `GroupOfIsometries`
- `submodule` – an invariant quotient module
- `is_left` – bool (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)
```
class sage.groups.matrix_gps.isometries.GroupActionOnSubmodule(MatrixGroup, submodule, is_left=False)

Bases: sage.categories.action.Action

Matrix group action on a submodule from the right.

INPUT:

- MatrixGroup – an instance of `GroupOfIsometries`
- submodule – an invariant submodule
- is_left – bool (default: False)

EXAMPLES:

```python
sage: from sage.groups.matrix_gps.isometries import GroupOfIsometries
sage: S = span(ZZ,[[0,1]])
```

```python
sage: g = Matrix(QQ,2,[1,0,0,-1])
```

```python
sage: G = GroupOfIsometries(2, ZZ, [g], invariant_bilinear_form=matrix.identity(2),
˓→invariant_submodule=S)
```

```python
sage: g = G.an_element()
```

```python
sage: x = S.an_element()
```

```python
sage: x*g
```

```
(0, -1)
```

```python
sage: (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: sage.groups.matrix_gps.finitely_generated.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 – a list of matrices over the base ring
- invariant_bilinear_form – a symmetric matrix
- category – (default: None) a category of groups
- check – bool (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:
```python
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
```

Infinite groups are O.K. too:

```python
sage: bil = Matrix(ZZ,4,[0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 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
```

**invariant_bilinear_form()**

Return the symmetric bilinear form preserved by the orthogonal group.

OUTPUT:

* the matrix defining the bilinear form

EXAMPLES:

```python
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]
```

### 25.12 Symplectic Linear Groups

**EXAMPLES:**

```python
sage: G = Sp(4,GF(7)); G
Symplectic Group of degree 4 over Finite Field of size 7
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
sage: G.order()
276595200
```

25.12. Symplectic Linear Groups 397
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 trac ticket #26028)

`sage.groups.matrix_gps.symplectic.Sp(n, R, var='a', invariant_form=None)`

Return the symplectic group.

The special linear group $\text{GL}(d, R)$ 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:**

- $n$ – a positive integer
- $R$ – ring or an integer; if an integer is specified, the corresponding finite field is used
- $\text{var}$ – (optional, default: 'a') variable used to represent generator of the finite field, if needed
- $\text{invariant\_form}$ – (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:**

```python
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

sage: Sp(3, GF(7))
Traceback (most recent call last):
...
ValueError: the degree must be even
```

Using the `invariant_form` option:

```python
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]
```
(continues on next page)
sage: pm = Permutation([2,1,4,3]).to_matrix()
sage: g = Sp4(pm); 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]
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

class sage.groups.matrix_gps.symplectic.SymplecticMatrixGroup_gap(degree, base_ring, special,
sage_name, latex_string,
gap_command_string, category=None)

groups.matrix_gps.named_group.NamedMatrixGroup_gap, sage.groups.matrix_gps.
finitely_generated.FinitelyGeneratedMatrixGroup_gap

Symplectic group in GAP.

EXAMPLES:

sage: Sp(2,4)
Symplectic Group of degree 2 over Finite Field in a of size 2^2

sage: latex(Sp(4,5))
\text{Sp}_{4}(\mathbb{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]
class sage.groups.matrix_gps.symplectic.SymplecticMatrixGroup_generic(degree, base_ring, special, sage_name, latex_string, category=None, invariant_form=None)

Bases: sage.groups.matrix_gps.named_group.NamedMatrixGroup_generic

Symplectic Group over arbitrary rings.

EXAMPLES:

```
\begin{verbatim}
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})
\end{verbatim}
```

```
\begin{verbatim}
sage: Sp4m = Sp(4,QQ, invariant_form=(0, 0, 1, 0, 0, 0, 2, -1, 0, 0, 0, 0, -2, 0, ...
\rightarrow 0)); Sp4m
Symplectic Group of degree 4 over Rational Field with respect to alternating_
\rightarrow 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)
\end{verbatim}
```

```
\begin{verbatim}
invariant_form()
  Return the quadratic form preserved by the symplectic group.
  OUTPUT:
  A matrix.
  EXAMPLES:
  
  sage: Sp(4, QQ).invariant_form()
  [ 0 0 1 0 ]
  [ 0 0 1 0 ]
  [ 0 -1 0 0 ]
  [-1 0 0 0 ]
```

25.13 Unitary Groups $GU(n, q)$ and $SU(n, q)$

These are $n \times n$ unitary matrices with entries in $GF(q^2)$.

EXAMPLES:

```python
sage: G = SU(3,5)
sage: G.order()
378000
sage: G
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
sage: G.gens()
([ 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
```

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 trac ticket #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 $F_q$ live over $F_{q^2}$. So $GU(n, q)$ for a prime power $q$ constructs the matrix group over the base ring $GF(q^4)$.

**Note:** This group is also available via `groups.matrix.GU()`.

INPUT:

- $n$ – a positive integer
- $R$ – ring or an integer; if an integer is specified, the corresponding finite field is used
- $var$ – (optional, 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:

Return the general unitary group.

EXAMPLES:

```sage
sage: G = GU(3, 7); G
General Unitary Group of degree 3 over Finite Field in a of size 7^2
sage: G.gens()
(,[ a 0 0] , [6*a 6 1]
 [ 0 1 0] , [ 6 6 0]
 [ 0 0 5*a], [ 1 0 0])
sage: GU(2, QQ)
General Unitary Group of degree 2 over Rational Field
sage: G = GU(3, 5, var='beta')
sage: G.base_ring()
Finite Field in beta of size 5^2
sage: G.gens()
(,[ 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
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
False
sage: G.invariant_form()
[1 0 0]
[0 1 0]
[0 0 1]
sage: Gm.invariant_form()
[ 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
True
[0 0 1]
[1 0 0]
[0 1 0]
sage: Gm(pm)
Traceback (most recent call last):
  ...
TypeError: matrix must be unitary with respect to the hermitian form
[ E(5) 0]
[E(5)^4 2 0]
```

(continues on next page)
sage: GU(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: GU(2,QQ, invariant_form=[[1,0],[2,0]])
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 $F_q$ live over $F_{q^2}$. 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:**
- $n$ – a positive integer
- $R$ – ring or an integer; if an integer is specified, the corresponding finite field is used
- `var` – (optional, 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:**
Return the special unitary group.

**EXAMPLES:**

```python
sage: SU(3,5)
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
sage: SU(3, GF(5))
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
```

Using the `invariant_form` option:

```python
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)
[0 0 1]
[1 0 0]
[0 1 0]
sage: Gm(pm)
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]
sage: SU(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

class sage.groups.matrix_gps.unitary.UnitaryMatrixGroup_gap(degree, base_ring, special,
sage_name, latex_string,
gap_command_string,
category=None)
Bases: sage.groups.matrix_gps.unitary.UnitaryMatrixGroup_generic, sage.
groups.matrix_gps.named_group.NamedMatrixGroup_gap, sage.groups.matrix_gps.
finitely_generated.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]
class sage.groups.matrix_gps.unitary.UnitaryMatrixGroup_generic(
deregree, base_ring, special,
sage_name, latex_string,
category=None,  
invariant_form=None)

Bases: sage.groups.matrix_gps.named_group.NamedMatrixGroup_generic

General Unitary Group over arbitrary rings.

EXAMPLES:

```
sage: G = GU(3, GF(7)); G
General Unitary Group of degree 3 over Finite Field in a of size 7^2
sage: latex(G)
\text{GU}_{3}(\Bold{F}_{7^{2}})
```

```
sage: G = SU(3, GF(5)); G
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
sage: latex(G)
\text{SU}_{3}(\Bold{F}_{5^{2}})
```

```
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 \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]
```

sage.groups.matrix_gps.unitary.finite_field_sqrt(ring)

Helper function.

INPUT:

A ring.

OUTPUT:

Integer q such that ring is the finite field with \(q^2\) elements.

EXAMPLES:
25.14 Heisenberg Group

AUTHORS:

• Hilder Vitor Lima Pereira (2017-08): initial version

class sage.groups.matrix_gps.heisenberg.HeisenbergGroup(n=1, R=0)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.groups.
matrix_gps.finitely_generated.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: $\mathbb{Z}$) the ring $R$ or a positive integer as a shorthand for the ring $\mathbb{Z}/R\mathbb{Z}$

EXAMPLES:

```python
sage: H = groups.matrix.Heisenberg(); H
Heisenberg group of degree 1 over Integer Ring
sage: 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]
)
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
```

REFERENCES:

• Wikipedia article Heisenberg_group
cardinality()
Return the order of self.

EXAMPLES:

```python
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
```

order()
Return the order of self.

EXAMPLES:

```python
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
```

25.15 Affine Groups

AUTHORS:

- Volker Braun: initial version

class sage.groups.affine_gps.affine_group.AffineGroup(degree, ring)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.groups.group.Group

An affine group.
The affine group \( \text{Aff}(A) \) (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 \( \text{Aff}(A_V) \) 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

\[
\text{Aff}(A_V) = GL(V) \ltimes V.
\]

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:
  - \text{degree} – An integer. The degree of the affine group, that is, the dimension of the affine space the group is acting on.
  - \text{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.
  - \text{var} – (default: 'a') Keyword argument to specify the finite field generator name in the case where \text{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]
[ 4  5  6 ]  [11]
[ 7  8  0 ]  [12]
sage: F([[1,2,3],[4,5,6],[7,8,0]], [10,11,12])
[ 1  2  3 ]  [10]
[ 4  5  6 ]  [11]
[ 7  8  0 ]  [12]
```

(continues on next page)
Instead of specifying the complete matrix/vector information, you can also create special group elements:

```sage
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]
```

Some additional ways to create affine groups:

```sage
sage: A = AffineSpace(2, GF(4, 'a')); A
Affine Space of dimension 2 over Finite Field in a of size 2^2
sage: G = AffineGroup(A); G
Affine Group of degree 2 over Finite Field in a of size 2^2
sage: G is AffineGroup(2,4) # shorthand
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
```

REFERENCES:

- Wikipedia article Affine_group

Element

alias of `sage.groups.affine_gps.group_element.AffineGroupElement`

cardinality()

Return the cardinality of self.

EXAMPLES:

```sage
sage: AffineGroup(6, GF(5)).cardinality()
172882428468750000000000000000
sage: AffineGroup(6, ZZ).cardinality()
+Infinity
```

degree()

Return the dimension of the affine space.

OUTPUT:

An integer.

EXAMPLES:
### 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:**

```python
sage: G = AffineGroup(3, GF(5))
sage: G.linear([1,2,3,4,5,6,7,8,0])
[1 2 3]
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:**

```python
sage: G = AffineGroup(3, GF(5))
sage: 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:**

```python
sage: G = AffineGroup(3, GF(5))
sage: 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: 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
```

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]
```

some_elements()

Return some elements.

EXAMPLES:

```
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],  
... ]  

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]  

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  

25.16 Euclidean Groups  

AUTHORS:  
• Volker Braun: initial version  

class sage.groups.affine_gps.euclidean_group.EuclideanGroup(degree, ring)  
Bases: sage.groups.affine_gps.affine_group.AffineGroup  
an 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 $x = (x_1, \ldots, x_n)$ to $(x_1, \ldots, x_n, 1)$.

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 – An 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
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]))
[ 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])
[ 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])
[ 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] x + [0]
[ 3/7 6/7 -2/7] x + [0]
sage: E3.reflection([4,5,6])
[ 45/77 -40/77 -48/77] x + [0]
[ -48/77 -60/77 5/77] x + [0]
sage: E3.translation([1,2,3])
[ 1 0 0] x + [1]
[ 0 1 0] x + [2]
[ 0 0 1] x + [3]
```

Some additional ways to create Euclidean groups:

```
sage: A = AffineSpace(2, GF(4, 'a')); A
Affine Space of dimension 2 over Finite Field in a of size 2^2
sage: G = EuclideanGroup(A); G
Euclidean Group of degree 2 over Finite Field in a of size 2^2
sage: G is EuclideanGroup(2,4) # shorthand
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
```

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] # random
[1 2 2 1] [0]
[2 2 2 2] x + [1]
[1 1 2 2] [2]
sage: G.random_element() in G
True
```
25.17 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])
 [7 8 0] [12]
 sage: G = AffineGroup(2, ZZ)
 sage: g = G([[1, 1], [0, 1]], [0, 1])
 sage: h = G([[1, 2], [0, 1]], [0, 1])
 sage: g*h
 [1 3] 
 x |-> [0 1] x + [1]
 sage: h*g
 [1 3] 
 x |-> [0 1] x + [1]
 sage: 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: sage.structure.element.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 - bool (default: True). Whether to convert A into the correct matrix space and b into the correct vector space.
• check - bool (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 \( x \mapsto A x + b \)

EXAMPLES:

```
 sage: G = AffineGroup(2, GF(3))
 sage: g = G.random_element()
 sage: type(g)
 <class 'sage.groups.affine_gps.affine_group.AffineGroup_with_category.element_class'>
```
Conversion from a matrix and a matrix group element:

```
sage: M = Matrix(4, 4, [0, 0, -1, 1, 0, 0, -1, 0, 1, 0, 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]
```

**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: g.A()
[1 2 3]
[4 5 6]
[7 8 0]
```

**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: g.b()
(10, 11, 12)
```

**inverse()**

Return the inverse group element.

**OUTPUT:**

Another affine group element.

**EXAMPLES:**
```python
sage: G = AffineGroup(2, GF(3))
sage: g = G([1,2,3,4], [5,6])
sage: g
[1 2] [2]
| -> [0 1] x + [0]
sage: ~g
[1 1] [1]
| -> [0 1] x + [0]
sage: g * g.inverse()
[1 0] [0]
| -> [0 1] x + [0]
sage: g * g.inverse() == g.inverse() * g == G(1)
True
```

`list()`

Return list representation of `self`.

**EXAMPLES:**

```python
sage: F = AffineGroup(3, QQ)
sage: g = F([1,2,3,4,5,6,7,8,0], [10,11,12])
```

```python
sage: g
[1 2 3] [3]
[0 1 0] [5]
```

```python
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]]
```

`matrix()`

Return the standard matrix representation of `self`.

**See also:**

- `AffineGroup.linear_space()`

**EXAMPLES:**

```python
sage: G = AffineGroup(3, GF(7))
sage: g = G([1,2,3,4,5,6,7,8,0], [10,11,12])
```

```python
sage: g
[1 2 3] [3]
[0 1 0] [5]
```

```python
g.matrix()
[[1 2 3|3]
[4 5 6|4]
[0 1 0|5]
[------+-]
```

(continues on next page)
Composition of affine group elements equals multiplication of the matrices:

```
sage: g1 = G.random_element()
sage: g2 = G.random_element()
sage: g1.matrix() * g2.matrix() == (g1*g2).matrix()
True
```
26.1 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: sage.groups.group.Group, sage.manifolds.differentiable.manifold.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. \( \mathbb{Q} \) or \( \mathbb{R} \)

• name – a 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 \( \text{chart\_exp1()} \) and \( \text{chart\_exp2()} \).

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
```

Giving a different name to the group:

```sage
G = L.lie_group('H'); G
```

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)
```

Lie group multiplication has the usual product syntax:
The identity element is given by \texttt{one()}:

\begin{verbatim}
sage: e = G.one(); e
exp(0)
sage: e*k == k and k*e == k
True
\end{verbatim}

The default coordinate system is exponential coordinates of the first kind:

\begin{verbatim}
sage: G.default_chart() == G.chart_exp1()
True
sage: G.chart_exp1()
Chart (G, (x_0, x_1, x_2))
\end{verbatim}

Changing the default coordinates to exponential coordinates of the second kind will change how elements are printed:

\begin{verbatim}
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)
\end{verbatim}

The frames of left- or right-invariant vector fields are created using \texttt{left_invariant_frame()} and \texttt{right_invariant_frame()}:

\begin{verbatim}
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
sage: X[0].display(exp1_frame)
X_0 = \frac{\partial}{\partial x_0} - \frac{1}{2}x_1 \frac{\partial}{\partial x_2}
sage: X[0].display(exp2_frame)
X_0 = \frac{\partial}{\partial y_0}
sage: X[1].display(exp1_frame)
X_1 = \frac{\partial}{\partial x_1} + \frac{1}{2}x_0 \frac{\partial}{\partial x_2}
sage: X[1].display(exp2_frame)
X_1 = \frac{\partial}{\partial y_1} + x_0 \frac{\partial}{\partial y_2}
\end{verbatim}

Defining a left translation by a generic point:

\begin{verbatim}
sage: a = var('a'); b = var('b'); c = var('c')
sage: g = G.point([a, b, c]); g
exp(a*p1 + b*q1 + c*z)
sage: L_g = G.left_translation(g); L_g
\end{verbatim}

(continues on next page)
Diffeomorphism of the Lie group $G$ of Heisenberg algebra of rank 1 over $\text{Rational Field}$:

```python
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)
```

Verifying the left-invariance of the left-invariant frame:

```python
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
```

An element of the Lie algebra can be extended to a left or right invariant vector field:

```python
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 $\text{Rational Field}$
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
```

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:

```python
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)
```

### class Element

**Bases:**

- `sage.manifolds.point.ManifoldPoint`
- `sage.structure.element.MultiplicativeGroupElement`

A base class for an element of a Lie group.

**EXAMPLES:**

Elements of the group are printed in the default exponential coordinates:
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())
```

adjoint\(g\)

Return the adjoint map as an automorphism of the Lie algebra of \(\text{self}\).

INPUT:

- \(g\) – an element of \(\text{self}\)

For a Lie group element \(g\), the adjoint map \(Ad_g\) is the map on the Lie algebra \(g\) given by the differential of the conjugation by \(g\) at the identity.

If the Lie algebra of \(\text{self}\) 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
```

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

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

chart_exp1()

Return the chart of exponential coordinates of the first kind.

Exponential coordinates of the first kind are

\[ \exp(x_1X_1 + \cdots + x_nX_n) \mapsto (x_1, \ldots, x_n). \]

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))

chart_exp2()

Return the chart of exponential coordinates of the second kind.

Exponential coordinates of the second kind are

\[ \exp(x_nX_n) \cdots \exp(x_1X_1) \mapsto (x_1, \ldots, x_n). \]

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))

conjugation(g)

Return the conjugation by \( g \) as an automorphism of \( \text{self} \).

The conjugation by \( g \) on a Lie group \( G \) is the map

\[ G \to G, \quad h \mapsto ghg^{-1}. \]

INPUT:

* \( g \) – an element of \( \text{self} \)
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 \rightarrow G
(x_0, x_1, x_2) \mapsto (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 \(\text{self}\)

EXAMPLES:

```
sage: L.<X,Y,Z> = LieAlgebra(QQ, 2, step=2)
sage: G = L.lie_group()
sage: G.exp(X)
\exp(X)
sage: G.exp(Y)
\exp(Y)
sage: G.exp(X + Y)
\exp(X + Y)
```

\(\text{gens}\)()

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))
```

\(\text{left_invariant_extension}(X, \text{name}=None)\)

Return the left-invariant vector field that has the value \(X\) at the identity.

INPUT:

- \(X\) – an element of the Lie algebra of \(\text{self}\)
- \(\text{name}\) – (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 \(\mathbb{Q}\)
\(\mathbb{Q}\-
\to\) Rational Field
sage: X.display(H.chart_exp1().frame())
p1 = \frac{\partial}{\partial x_0} - \frac{1}{2}x_1 \frac{\partial}{\partial 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 \(\mathbb{Q}\)
\(\mathbb{Q}\-
\to\) 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 \(\mathbb{Q}\)
\(\mathbb{Q}\-
\to\) 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 = \frac{\partial}{\partial x_1} - \frac{1}{2}x_2 \frac{\partial}{\partial x_12}
sage: livf[1].display(coord_frame)
X_2 = \frac{\partial}{\partial x_2} + \frac{1}{2}x_1 \frac{\partial}{\partial x_12}
sage: livf[2].display(coord_frame)
X_12 = \frac{\partial}{\partial 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))

**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

\[ G \to G, \quad h \mapsto gh. \]

**INPUT:**
• g – an element of self

EXAMPLES:

A left translation in the Heisenberg group:

```python
sage: H = lie_algebras.Heisenberg(QQ, 1)
sage: p,q,z = H.basis()
sage: G = H.lie_group()
sage: g = G.exp(p)
```

Diffeomorphism of the Lie group G of Heisenberg algebra of rank 1 over Rational Field

```python
sage: L_g = G.left_translation(g); L_g
```

```
Diffeomorphism of the Lie group G of Heisenberg algebra of rank 1 over Rational Field
```

```python
sage: L_g.display(chart1=G.chart_exp1(), chart2=G.chart_exp1())
```

```
G \rightarrow G
(x_0, x_1, x_2) \mapsto (x_0 + 1, x_1, 1/2 \cdot x_1 + x_2)
```

Left translation by a generic element:

```python
sage: h = G.point([var('a'), var('b'), var('c')])
sage: L_h = G.left_translation(h)
```

```python
sage: L_h.display(chart1=G.chart_exp1(), chart2=G.chart_exp1())
```

```
G \rightarrow G
(x_0, x_1, x_2) \mapsto (a + x_0, b + x_1, -1/2 \cdot b \cdot x_0 + 1/2 \cdot a \cdot x_1 + c + x_2)
```

`lie_algebra()`

Return the Lie algebra of self.

EXAMPLES:

```python
sage: L = LieAlgebra(QQ, 2, step=2)
sage: G = L.lie_group()
sage: 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:

```python
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 \cdot x_2 \partial/∂x_12
sage: livf[1].display(coord_frame)
X_2 = ∂/∂x_2 + 1/2 \cdot x_1 \partial/∂x_12
sage: livf[2].display(coord_frame)
X_12 = ∂/∂x_12
```

Examples of custom labeling for the frame:
\texttt{sage: G.left_invariant_frame(symbol='Y')}
Vector frame \((G, (Y_1,Y_2,Y_{12}))\)

\texttt{sage: G.left_invariant_frame(symbol='Z', indices=None)}
Vector frame \((G, (Z_0,Z_1,Z_2))\)

\texttt{sage: G.left_invariant_frame(symbol='\mathcal{W}', indices=('a','b','c'))}
Vector frame \((G, (\mathcal{W}_a,\mathcal{W}_b,\mathcal{W}_c))\)

\texttt{log(x)}

Return the logarithm of the element \(x\) of \texttt{self}.

INPUT:

- \(x\) – an element of \texttt{self}

The logarithm is by definition the inverse of \texttt{exp()}. If the Lie algebra of \texttt{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:

\begin{verbatim}
\texttt{sage: L.<X,Y,Z> = LieAlgebra(QQ, 2, step=2)
\texttt{sage: G = L.lie_group()}
\texttt{sage: G.log(G.exp(X)) == X}
True
\texttt{sage: G.log(G.exp(X)*G.exp(Y))}
X + Y + 1/2*Z
\end{verbatim}

The logarithm is not defined for abstract (symbolic) points:

\begin{verbatim}
\texttt{sage: g = G.point([var('a'), 1, 2]); g}
exp(a*X + Y + 2*Z)
\texttt{sage: G.log(g)}
Traceback (most recent call last):
... TypeError: unable to convert a to a rational
\end{verbatim}

\texttt{one()}

Return the identity element of \texttt{self}.

EXAMPLES:

\begin{verbatim}
\texttt{sage: L = LieAlgebra(QQ, 2, step=4)
\texttt{sage: G = L.lie_group()}
\texttt{sage: G.one()}
exp(0)
\end{verbatim}

\texttt{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 \texttt{self}
- \texttt{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 = \partial/\partial x_0 + 1/2*x_1 \partial/\partial 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
```

```
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 = \partial/\partial x_1 + 1/2*x_2 \partial/\partial x_12
sage: rivf[1].display(coord_frame)
XR_2 = \partial/\partial x_2 - 1/2*x_1 \partial/\partial x_12
sage: rivf[2].display(coord_frame)
XR_12 = \partial/\partial 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))
```

```
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

$$G \to G, \quad h \mapsto hg.$$ 

INPUT:

• $g$ – an element of self

EXAMPLES:

A right translation in the Heisenberg group:

```python
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 \to G
(x_0, x_1, x_2) \mapsto (x_0 + 1, x_1, -1/2*x_1 + x_2)
```

Right translation by a generic element:

```python
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 \to G
(x_0, x_1, x_2) \mapsto (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:

```python
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 = \frac{\partial}{\partial x_1} + \frac{1}{2}*x_2 \frac{\partial}{\partial x_12}
sage: rivf[1].display(coord_frame)
XR_2 = \frac{\partial}{\partial x_2} - \frac{1}{2}*x_1 \frac{\partial}{\partial x_12}
sage: rivf[2].display(coord_frame)
XR_12 = \frac{\partial}{\partial x_12}
```

Examples of custom labeling for the frame:

```python
sage: G.right_invariant_frame(symbol='Y')
Vector frame (G, (Y_1,Y_2,Y_12))
```

(continues on next page)
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))

step()
Return the nilpotency step of self.

EXAMPLES:

sage: L = LieAlgebra(QQ, 2, step=4)
sage: G = L.lie_group()
sage: G.step()
4
27.1 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 \texttt{mem\_err} variable as input should make use of this variable.

D. \texttt{generate\_children}:

\textbf{Signature:}

\begin{verbatim}
int generate_children(void *S, aut_gp_and_can_lab *group, iterator *it)
\end{verbatim}

This function receives a pointer to an iterator \texttt{it}. The iterator has two fields: \texttt{data} and \texttt{next}. The function \texttt{generate\_children} should set these two fields, returning 1 to indicate a memory error, or 0 for no error.

The function that \texttt{next} points to takes \texttt{data} as an argument, and should return a (\texttt{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 \( \gamma \) 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 \( \gamma \) is appropriately extended if necessary to operate on \( C_1 \) and \( C_2 \). It is essential for this iterator to handle its own \texttt{data}. If the \texttt{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 \texttt{data} variable should be cleared by the \texttt{next} function, because the iterator struct itself will be deallocated.

The \texttt{next} function must check \texttt{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 \texttt{mem\_err} attribute of the \texttt{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. \texttt{apply\_augmentation}:

\textbf{Signature:}

\begin{verbatim}
void *apply_augmentation(void *parent, void *aug, void *child, int *degree, bint *mem_err)
\end{verbatim}

This function takes the \texttt{parent}, applies the augmentation \texttt{aug} and returns a pointer to the corresponding child object (freeing \texttt{aug} if necessary). Should also update \texttt{degree[0]} to be the degree of the new child.

F. \texttt{free\_object}:

\textbf{Signature:}

\begin{verbatim}
void free_object(void *child)
\end{verbatim}

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 \texttt{child}.

G. \texttt{free\_iter\_data}:

\textbf{Signature:}

\begin{verbatim}
void free_iter_data(void *data)
\end{verbatim}

This function deallocates the data part of the iterator which is set up by \texttt{generate\_children}.

H. \texttt{free\_aug}:

\textbf{Signature:}

\begin{verbatim}
void free_aug(void *aug)
\end{verbatim}
This function frees an augmentation as generated by the iterator returned by \texttt{generate\_children}.

\textbf{I. canonical\_parent:}

\textbf{Signature:}

\begin{verbatim}
void *canonical_parent(void *child, void *parent, int *permutation, int *degree, bint *mem_err)
\end{verbatim}

Apply the \texttt{permutation} to the \texttt{child}, determine an arbitrary but fixed parent, apply the inverse of \texttt{permutation} to that parent, and return the resulting object. Must also set the integer \texttt{degree} points to the degree of the returned object.

\textbf{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.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: import sage.groups.perm_gps.partn_ref.canonical_augmentation
\end{verbatim}

\textbf{REFERENCE:}


\section*{27.2 Data structures}

This module implements basic data structures essential to the rest of the partn\_ref module.

\textbf{REFERENCES:}


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 \texttt{perms} in \texttt{L} of degree \texttt{n} is of the correct order, by comparing with \texttt{GAP}. Don't test if the group is of size greater than limit.
27.3 Graph-theoretic partition backtrack functions

EXAMPLES:

```
sage: import sage.groups.perm_gps.partn_ref.refinement_graphs
```

REFERENCE:


```python
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:

```python
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("{} {}\n".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):
    ```python
```sage: from sage.groups.perm_gps.partn_ref.refinement_graphs import coarsest_equitable_refinement
sage: from sage.graphs.base.sparse_graph import SparseGraph

(continues on next page)
sage: coarsest_equitable_refinement(SparseGraph(7), [[0], [1, 2, 3, 4], [5, 6], [0], [1, 2, 3, 4], [5, 6]])

```python
sage: 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
```

```python
sage: for n in [0..7]:
....:     print(generate_dense_graphs_edge_addition(n, 0))
1
1
2
4
11
34
156
1044
```

```python
sage: generate_dense_graphs_edge_addition(8, 0) # long time - about 14 seconds at 2.4 GHz
12346
```

```python
sage: from sage.groups.perm_gps.partn_ref.refinement_graphs import generate_dense_graphs_vert_addition
sage: for n in [0..7]:
....:     print(generate_dense_graphs_vert_addition(n, 0))
1
1
2
4
11
34
156
1044
```

EXAMPlES:

```python
sage: 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

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]]

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: 2, 2: 1}
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}
sage.groups.perm_gps.partn_ref.refinement_graphs.orbit_partition(gamma, list_perm=False)

Assuming that $G$ is a graph on vertices $0, 1, \ldots, n-1$, 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 $1, 2, \ldots, n$ 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 $i \mapsto$

**EXAMPLES:**

```python
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]]
```

sage.groups.perm_gps.partn_ref.refinement_graphs.random_tests(num=10, n_max=60, perms_per_graph=5)

Tests 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` – a 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 \(G\) 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
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

Graphs on zero vertices:

```sage```
G = Graph()
```sage```

g(G, [[]], order=True)

([], Graph on 0 vertices, 1)
```sage```

Graphs on one vertex:

```sage```
G = Graph(1)
```sage```

g(G, [[0]], order=True)

([], Graph on 1 vertex, 1)
```sage```

Graphs on two vertices:

```sage```
G = Graph(2)
```sage```

g(G, [[0,1]], order=True)

([[1, 0]], Graph on 2 vertices, 2)
```sage```

g(G, [[0],[1]], order=True)

([], Graph on 2 vertices, 1)
```sage```

g.add_edge(0,1)
```sage```

g(G, [[0,1]], order=True)

(continues on next page)
Graphs on three vertices:

```python
sage: G = Graph(3)
```

```python
sage: st(G, [[0,1,2]], order=True)
([0, 1, 2], Graph on 3 vertices, 6)
```

```python
sage: st(G, [[0],[1,2]], order=True)
([], Graph on 3 vertices, 2)
```

```python
sage: st(G, [[0,1],[2]], order=True)
([0, 1, 2], Graph on 3 vertices, 2)
```

The Dodecahedron has automorphism group of size 120:

```python
sage: G = graphs.DodecahedralGraph()
sage: Pi = [range(20)]
sage: st(G, Pi, order=True)[2]
120
```

The three-cube has automorphism group of size 48:

```python
sage: G = graphs.CubeGraph(3)
```

```python
sage: G.relabel()
```

```python
sage: Pi = [G.vertices()]
```

```python
sage: st(G, Pi, order=True)[2]
48
```

We obtain the same output using different types of Sage graphs:

```python
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):
```

(continues on next page)
bsg.add_edge(i, j)
if bde.has_arc(i, j):
bdg.add_edge(i, j)

sage: a, b.graph6_string()

sage: a == asp
True
sage: a == ade
True
sage: b == bsg
True
sage: b == bdg
True

Cubes!:

sage: C = graphs.CubeGraph(1)
sage: gens, order = st(C, [C.vertices()], lab=False, order=True); order
2
sage: C = graphs.CubeGraph(2)
sage: gens, order = st(C, [C.vertices()], lab=False, order=True); order
8
sage: C = graphs.CubeGraph(3)
sage: gens, order = st(C, [C.vertices()], lab=False, order=True); order
48
sage: C = graphs.CubeGraph(4)
sage: gens, order = st(C, [C.vertices()], lab=False, order=True); order
384
sage: C = graphs.CubeGraph(5)
sage: gens, order = st(C, [C.vertices()], lab=False, order=True); order
3840
sage: C = graphs.CubeGraph(6)
sage: gens, order = st(C, [C.vertices()], 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()], dig=True, use_indicator_function=False)
sage: c,d = st(D2, [D2.vertices()], dig=True, use_indicator_function=False)
sage: 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()
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(':Dkw')
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]]
```

### 27.4 Partition backtrack functions for lists – a simple example of using partn_ref

**EXAMPLES:**

```
sage: 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]
```
27.5 Partition backtrack functions for matrices

EXAMPLES:

```python
sage: import sage.groups.perm_gps.partn_ref.refinement_matrices
```

REFERENCE:


class sage.groups.perm_gps.partn_ref.refinement_matrices.MatrixStruct

    Bases: object

    automorphism_group()
    
    Returns 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:

    ```python
    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])
    ```

    canonical_relabeling()
    
    Returns a canonical relabeling (in list permutation format).
    
    For more examples, see self.run().

    EXAMPLES:

    ```python
    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]
    ```

    display()
    
    Display the matrix, and associated data.
    
    EXAMPLES:

    ```python
    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
    ```

(continues on next page)
is_isomorphic(other)
Calculate whether self is isomorphic to other.

EXAMPLES:

```python
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]
```

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:**

```python
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()
```

```python
sage: M.automorphism_group()
([0, 2, 1], 2, [1])
```

```python
sage: M.canonical_relabeling()
[0, 1, 2]
```

```python
sage: M = MatrixStruct(matrix(GF(3),[[0,1,2],[0,2,1],[1,0,2],[1,2,0],[2,0,1],[2,1,0]]))
```

```python
sage: M.automorphism_group()[1] == 6
True
```

```python
sage: M = MatrixStruct(matrix(GF(3),[[0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,2]]))
```

```python
sage: M.automorphism_group()[1] == factorial(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))

Tests 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.

27.5. Partition backtrack functions for matrices
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.
28.1 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:**

```python
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: G.is_finite()
True
sage: G.conjugacy_classes_representatives()
([1 0] [0 2] [0 1] [2 0] [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])
```

**class** `sage.groups.matrix_gps.named_group.NamedMatrixGroup_gap(degree, base_ring, special, sage_name, latex_string, gap_command_string, category=None)`

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:**

```python
sage: G = GL(2, GF(3))
sage: from sage.groups.matrix_gps.named_group import NamedMatrixGroup_gap
sage: isinstance(G, NamedMatrixGroup_gap)
True
```

```python
sage: G = GL(2, QQ)
sage: from sage.groups.matrix_gps.named_group import NamedMatrixGroup_generic
sage: isinstance(G, NamedMatrixGroup_generic)
True
```

See also:

See the examples for `GU()`, `SU()`, `Sp()`, etc. as well.

```python
sage: from sage.groups.matrix_gps.named_group import normalize_args_invariant_form
sage: 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 $\mathbb{R}$ 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 trac ticket #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)`.
INDICES AND TABLES

- Index
- Module Index
- Search Page
BIBLIOGRAPHY

g

sage.groups.abelian_gps.abelian_aut, 195
sage.groups.abelian_gps.abelian_group, 171
sage.groups.abelian_gps.abelian_group_element, 211
sage.groups.abelian_gps.abelian_group_gap, 187
sage.groups.abelian_gps.abelian_group_morphism, 215
sage.groups.abelian_gps.dual_abelian_group, 205
sage.groups.abelian_gps.dual_abelian_group_element, 213
sage.groups.abelian_gps.element_base, 208
sage.groups.abelian_gps.values, 200
sage.groups.additive_abelian.additive_abelian_group, 217
sage.groups.additive_abelian.additive_abelian_wrapper, 221
sage.groups.affine_gps.affine_group, 407
sage.groups.affine_gps.euclidean_group, 412
sage.groups.affine_gps.group_element, 415
sage.groups.braid, 83
sage.groups.class_function, 155
sage.groups.conjugacy_classes, 167
sage.groups.cubic_braid, 111
sage.groups.finitely_presented, 59
sage.groups.finitely_presented_named, 79
sage.groups.free_group, 51
sage.groups.generic, 37
sage.groups.group, 3
sage.groups.group_exp, 135
sage.groups.group_semidirect_product, 139
sage.groups.groups_catalog, 1
sage.groups.indexed_free_group, 125
sage.groups.libgap_group, 21
sage.groups.libgap_mixin, 23
sage.groups.libgap_morphism, 7
sage.groups.libgap_wrapper, 13
sage.groups.lie_gps.nilpotent_lie_group, 419
sage.groups.matrix_gps.binary_dihedral, 374
sage.groups.matrix_gps.catalog, 349
sage.groups.matrix_gps.coxeter_group, 375
sage.groups.matrix_gps.finitely_generated, 360
sage.groups.matrix_gps.group_element, 354
sage.groups.matrix_gps.heisenberg, 406
sage.groups.matrix_gps.homset, 373
sage.groups.matrix_gps.isometries, 395
sage.groups.matrix_gps.linear, 383
sage.groups.matrix_gps.matrix_group, 349
sage.groups.matrix_gps.morphism, 373
sage.groups.matrix_gps.named_group, 445
sage.groups.matrix_gps.orthogonal, 386
sage.groups.matrix_gps.symplectic, 397
sage.groups.matrix_gps.unitary, 401
sage.groups.misc_gps.argument_groups, 223
sage.groups.misc_gps.imaginary_groups, 229
sage.groups.misc_gps.misc_groups, 145
sage.groups.pari_group, 35
sage.groups.perm_gps.constructor, 233
sage.groups.perm_gps.cubegroup, 331
sage.groups.perm_gps.partn_ref.canonical_augmentation, 431
sage.groups.perm_gps.partn_ref.data_structures, 433
sage.groups.perm_gps.partn_ref.refinement_graphs, 434
sage.groups.perm_gps.partn_ref.refinement_lists, 441
sage.groups.perm_gps.partn_ref.refinement_matrices, 442
sage.groups.perm_gps.permgroup, 236
sage.groups.perm_gps.permgroup_element, 319
sage.groups.perm_gps.permgroup_morphism, 328
sage.groups.perm_gps.permgroup_named, 288
sage.groups.perm_gps.permutation_groups_catalog, 233
sage.groups.perm_gps.symgp_conjugacy_class, 345
sage.groups.raag, 129
sage.groups.seminomomial_transformations.seminomomial_transformation, 151
sage.groups.seminomomial_transformations.seminomomial_transformations, 444
INDEX

A
A() (sage.groups.affine_gps.group_element.AffineGroupElement method), 416
abelian_invariants() (sage.groups.finitely_presented.FinitelyPresentedGroup method), 61
abelian_invariants() (sage.groups.free_group.FreeGroup_class method), 55
AbelianGroup (class in sage.groups.group), 3
AbelianGroup() (in module sage.groups.abelian_gps.abelian_group), 173
AbelianGroup_class (class in sage.groups.abelian_gps.abelian_group), 174
AbelianGroup_gap (class in sage.groups.abelian_gps.abelian_group_gap), 192
AbelianGroup_subgroup (class in sage.groups.abelian_gps.abelian_group), 184
AbelianGroupAutomorphism (class in sage.groups.abelian_gps.abelian_aut), 196
AbelianGroupAutomorphismGroup (class in sage.groups.abelian_gps.abelian_aut), 197
AbelianGroupAutomorphismGroup_gap (class in sage.groups.abelian_gps.abelian_aut), 198
AbelianGroupAutomorphismGroup_subgroup (class in sage.groups.abelian_gps.abelian_aut), 199
AbelianGroupElement (class in sage.groups.abelian_gps.abelian_group_element), 212
AbelianGroupElement_gap (class in sage.groups.abelian_gps.abelian_group_gap), 187
AbelianGroupElement_polycyclic (class in sage.groups.abelian_gps.abelian_group_gap), 188
AbelianGroupElementBase (class in sage.groups.abelian_gps.element_base), 208
AbelianGroupAutomorphismGroup_gap (class in sage.groups.abelian_gps.abelian_group_gap), 199
AbelianGroupAutomorphismGroup_subgroup (class in sage.groups.abelian_gps.abelian_group_gap), 190
AbelianGroupWithValues() (in module sage.groups.abelian_gps.values), 200
AbelianGroupWithValues_class (class in sage.groups.abelian_gps.values), 203
AbelianGroupWithValuesElement (class in sage.groups.abelian_gps.values), 201
AbelianGroupWithValuesEmbedding (class in sage.groups.abelian_gps.values), 202
absolute_length() (sage.groups.perm_gps.permgroup_element.SymmetricGroupElement method), 327
AbstractArgument (class in sage.groups.misc_gps.argument_groups), 224
AbstractArgumentGroup (class in sage.groups.misc_gps.argument_groups), 224
act_to_right() (sage.groups.group_semidirect_product.GroupSemidirectProduct method), 140
action_on_root_indices() (sage.groups.matrix_gps.coxeter_group.CoxeterMatrixGroup.Element method), 377
adams_operation() (sage.groups.class_function.ClassFunction_gap method), 156
adams_operation() (sage.groups.class_function.ClassFunction_libgap method), 160
add_strings() (in module sage.groups.abelian_gps.dual_abelian_group_element), 455
Index
chart_exp2() (sage.groups.lie_gps.nilpotent_lie_group.NilpotentLieGroup method), 423

class_function() (sage.groups.libgap_mixin.GroupMiximLibGAP method), 25

ClassFunction() (in module sage.groups.class_function), 155

ClassFunction_gap (class in sage.groups.class_function), 160

classical_invariant_form() (sage.groups.cubic_braid.CubicBraidGroup method), 122

color_of_square() (in module sage.groups.perm_gps.cubegroup), 342

tCommutator() (sage.groups.braid.Braid method), 91

ComplexReflectionGroup (class in sage.groups.perm_gps.permgroup_named), 247

components_in_closure() (sage.groups.braid.Braid method), 92

cocent_equitable_refinement() (in module sage.groups.perm_gps.partn_ref.refinement_graph), 434

codcogrees() (sage.groups.perm_gps.permgroup_named.ComplexReflectionGroup method), 291

Cohomology() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 246

Cohomology() (sage.groups.raag.RightAngledArtinGroup method), 132

color_of_square() (in module sage.groups.perm_gps.cubegroup), 342

colored_jones_polynomial() (sage.groups.braid.Braid method), 91

conjugacy_class() (sage.groups.libgap_mixin.GroupMiximLibGAP method), 26

conjugacy_class() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 249

conjugacy_class() (sage.groups.perm_gps.permgroup_named.SymmetricGroup method), 313

conjugacy_classes_iterator() (sage.groups.perm_gps.permgroup_named.SymmetricGroup method), 314

conjugacy_classes_representatives() (sage.groups.libgap_mixin.GroupMiximLibGAP method), 26

conjugacy_classes_representatives() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 250

coarsest_equitable_refinement() (in module sage.groups.perm_gps.partn_ref.refinement_graph), 434

codcogrees() (sage.groups.perm_gps.permgroup_permgroup.Named method), 291

CohomologyRAAG (class in sage.groups.raag), 129

CohomologyRAAG.Element (class in sage.groups.raag), 129

color_of_square() (in module sage.groups.perm_gps.cubegroup), 342

coloed Jones polynomial() (sage.groups.braid.Braid method), 91

conjugacy_class() (sage.groups.libgap_mixin.GroupMiximLibGAP method), 26

conjugacy_class() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 249

conjugacy_class() (sage.groups.perm_gps.permgroup_permgroup.Named method), 313

conjugacy_class_iterator() (in module sage.groups.perm_gps.symgp_conjugacy_class), 346

conjugacy_classes() (sage.groups.libgap_mixin.GroupMiximLibGAP method), 26

conjugacy_classes() (sage.groups.perm_gps.permgroup_permgroup.Named method), 313

conjugacy_classes_representatives() (sage.groups.libgap_mixin.GroupMiximLibGAP method), 26

conjugacy_classes_representatives() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 250

ConjugacyClass (class in sage.groups.conjugacy_classes), 167

ConjugacyClassGAP (class in sage.groups.conjugacy_classes), 169

conjugate() (sage.groups.libgap_mixin.GroupMiximLibGAP method), 26

conjugating_braid() (sage.groups.braid.Braid method), 92

conjugation() (sage.groups.lib_gps.nilpotent_lie_group.Ni1potentLieGroup method), 423

construction() (sage.groups.group_semidirect_product.GroupSemidirectProduct method), 141

construction() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 252

cosets() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 253

cover() (sage.groups.abelian_gps.abelian_group_gap.AbelianGroupQuotient_gap method), 189

cover_and_relations_from_invariants() (in module sage.groups.additive_abelian.additive_abelian_group), 221

cofield_matrix_ring() (sage.groups.abelian_gps.abelian_gap.AbelianGroupAutomorphism method), 221

coxeter_matrix() (sage.groups.matrix_gps.coxeter_group.CoxeterMatrixGroup method), 314

coxeter_matrix() (sage.groups.perm_gps.permgroup_named.SymmetricGroup method), 314

CoxeterMatrixGroup (class in sage.groups.matrix_gps.coxeter_group), 375

CoxeterMatrixGroup.Element (class in sage.groups.matrix_gps.coxeter_group), 377
create_key_and_extra_args()
create_object()
cubic_braid_subgroup()
CubicBraidElement (class in sage.groups.perm_gps.cubegroup), 332
CubicBraidGroup (class in sage.groups.cubic_braid), 114
CubicBraidGroup.type (class in sage.groups.cubic_braid), 124
cubie() (sage.groups.perm_gps.cubegroup.CubeGroup class in module sage.groups.perm_gps.cubegroup), 342
cubie_colors() (in module sage.groups.perm_gps.cubegroup), 342
cubie_faces() (in module sage.groups.perm_gps.cubegroup), 342
cycle_string() (sage.groups.perm_gps.permgroup_element.PermutationGroupElement class in module sage.groups.perm_gps.permgroup_element), 321
cycle_tuples() (sage.groups.perm_gps.permgroup_element.PermutationGroupElement class in module sage.groups.perm_gps.permgroup_element), 321
cycle_type() (sage.groups.perm_gps.permgroup_element.PermutationGroupElement class in module sage.groups.perm_gps.permgroup_element), 322
cycles() (sage.groups.perm_gps.permgroup_element.PermutationGroupElement method), 322
CyclicPermutationGroup (class in sage.groups.perm_gps.permgroup_named), 293
CyclicPresentation() (in module sage.groups.finitely_presented_named), 80
degree() (sage.groups.class_function.ClassFunction_group_method), 156
deprecated() (sage.groups.class_function.ClassFunction_libgap_method), 353
degree() (sage.groups.class_function.ClassFunction_libgap_group_method), 114
degree() (sage.groups.perm_gps.permgroup.PermutationGroup_element_method), 254
degree() (sage.groups.perm_gps.permgroup_permgroup_named_TransitiveGroup_method), 317
degree() (sage.groups.semimonomial_transformations.semimonomial_triang_method), 149
degree_on_basis() (sage.groups.raag.CohomologyRAAG_method), 129
degrees() (sage.groups.perm_gps.permgroup_named.ComplexReflectionGroup_method), 292
derived_series() (sage.groups.perm_gps.permgroup.PermutationGroup_element_class in module sage.groups.perm_gps.permgroup_element), 255
descents() (sage.groups.matrix_gps.coxeter_group.CoxeterMatrixGroupElement_method), 378
determinant_character() (sage.groups.class_function.ClassFunction_gap_method), 157
determinant_character() (sage.groups.class_function.ClassFunction_libgap_method), 161
direct_product() (sage.groups.finitely_presented.FinitelyPresentedGroup_method), 64
direct_product() (sage.groups.perm_gps.permgroup.PermutationGroup_element_method), 255
dimension_of_TL_space() (sage.groups.braid.BraidGroup_class_method), 105
direct_product() (sage.groups.finitely_presented.FinitelyPresentedGroup_element_class in module sage.groups.finitely_presented_named), 80
DihedralGroup (class in module sage.groups.finitely_presented_named), 296
DihedralPresentation() (in module sage.groups.finitely_presented_named), 81
direct_product() (sage.groups.finitely_presented.FinitelyPresentedGroup_element_class in module sage.groups.finitely_presented_named), 64
direct_product() (sage.groups.perm_gps.permgroup.PermutationGroup_element_class in module sage.groups.perm_gps.permgroup_named), 287
discrete_log() (in module sage.groups.generic), 40
discrete_log() (in module sage.groups.generic), 40
discrete_log() (in module sage.groups.generic), 40
discrete_log() (in module sage.groups.generic), 40
F
F() (sage.groups.perm_gps.cubegroup.CubeGroup method), 333
faces() (sage.groups.perm_gps.cubegroup.CubeGroup method), 334
facets() (sage.groups.perm_gps.cubegroup.CubeGroup method), 335
facets() (sage.groups.perm_gps.cubegroup.RubiksCube method), 340
field_of_definition() (sage.groups.perm_gps.permgroup_named.PermutationGroup method), 304
finite_field_sqrt() (in module sage.groups.matrix_gps.unitary), 405
FiniteGroup (class in sage.groups.group), 3
finitely_presented_group() (sage.groups.finitely_presented.RewritingSystem method), 73
FinitelyGeneratedAbelianPresentation() (in module sage.groups.finitely_presented_named), 81
FinitelyGeneratedHeisenbergPresentation() (in module sage.groups.finitely_presented_named), 82
FinitelyGeneratedMatrixGroup_gap (class in sage.groups.matrix_gps.finitely_generated), 361
FinitelyGeneratedMatrixGroup_generic (class in sage.groups.matrix_gps.finitely_generated), 369

G
gap() (sage.groups.class_function.ClassFunction_libgap method), 162
gap() (sage.groups.finitely_presented.RewritingSystem method), 74
gap() (sage.groups.libgap_morphism.GroupMorphism_libgap method), 9
gap() (sage.groups.libgap_wrapper.ElementLibGAP method), 14
gap() (sage.groups.libgap_wrapper.ParentLibGAP method), 17
gap() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 257
gap() (sage.groups.perm_gps.permgroup_element.PermutationGroupElement method), 323
gcd() (sage.groups.braid.Braid method), 94
gen() (sage.groups.abelian_gps.abelian_group.AbelianGroup_class method), 177
gen() (sage.groups.abelian_gps.abelian_group.AbelianGroup_subgroup method), 185
gen() (sage.groups.abelian_gps.dual_abelian_group.DualAbelianGroup_class method), 206
GroupHomset_libgap (class in sage.groups.libgap_morphism), 7
GroupLibGAP (class in sage.groups.libgap_group), 21
GroupMixinLibGAP (class in sage.groups.libgroup.mixin), 23
GroupMorphism_libgap (class in sage.groups.libgap_morphism), 8
GroupMorphismWithGensImages (class in sage.groups.finitely_presented), 72
GroupOfIsometries (class in sage.groups.matrix_gps.isometries), 396
GroupSemidirectProduct (class in sage.groups.group_semidirect_product), 139
GroupSemidirectProductElement (class in sage.groups.group_semidirect_product), 142
GU() (in module sage.groups.matrix_gps.unitary), 401

hap_decorator() (in module sage.groups.perm_gps.permgroup), 288
has_descent() (sage.groups.perm_gps.permgroup_element.PermutationGroupGeneric, method), 324
has_element() (sage.groups.perm_gps.permgroup.PermutationGroup_generic, method), 260
has_left_descent() (sage.groups.perm_gps.permgroup_element.PermutationGroupElement, method), 327
has_regular_subgroup() (sage.groups.perm_gps.permgroup.PermutationGroup_permgroup, method), 260
has_right_descent() (sage.groups.matrix_gps.coxeter_group.CoxeterMatrixGroup_element, method), 379
HeisenbergGroup (class in sage.groups.matrix_gps.heisenberg), 406
holomorph() (sage.groups.perm_gps.permgroup.PermutationGroup_generic, method), 260
homology() (sage.groups.perm_gps.permgroup.PermutationGroup_permgroup, method), 261
homology_part() (sage.groups.perm_gps.permgroup.PermutationGroup_permgroup, method), 262

identity() (sage.groups.perm_gps.permgroup.PermutationGroup_generic, method), 262
image() (sage.groups.misc_gps.imaginary_groups.ImaginaryElement, method), 230
image() (sage.groups.abelian_gps.abelian_group_morphism.AbelianGroupMorphism, method), 216
index2singmaster() (in module sage.groups.perm_gps.cubegroup), 343
index_set() (sage.groups.perm_gps.permgroup_named.ComplexReflectionGroup, method), 292
index_set() (sage.groups.perm_gps.permgroup_named.SymmetricGroup, method), 314

IndexedFreeAbelianGroup (class in sage.groups.indexed_free_group), 125
IndexedFreeAbelianGroup.Element (class in sage.groups.indexed_free_group), 125
IndexedFreeGroup (class in sage.groups.indexed_free_group), 125
IndexedFreeGroup.Element (class in sage.groups.indexed_free_group), 126

indexed() (sage.groups.class_function.ClassFunction_gap, method), 57
induct() (sage.groups.class_function.ClassFunction_gap, method), 162
intersection() (sage.groups.libgap_mixin.GroupMixinLibGAP, method), 27
intersection() (sage.groups.perm_gps.permgroup.PermutationGroup_permgroup, method), 263
inv_list() (in module sage.groups.perm_gps.permgroup), 343
invariant_bilinear_form() (sage.groups.matrix_gps.isometries.GroupOfIsometries, method), 397
invariant_bilinear_form() (sage.groups.matrix_gps.orthogonal.OrthogonalMatrixGroup_gap, method), 389
invariant_bilinear_form() (sage.groups.matrix_gps.orthogonal.OrthogonalMatrixGroup_gap, method), 391
invariant_form() (sage.groups.matrix_gps.orthogonal.OrthogonalMatrixGroup_gap, method), 389
invariant_form() (sage.groups.matrix_gps.orthogonal.OrthogonalMatrixGroup_gap, method), 391

Index
invariant_form() (sage.groups.matrix_gps.symplectic.SymplecticMatrixGroup_gap method), 399
invariant_form() (sage.groups.matrix_gps.unitary.UnitaryMatrixGroup_gap method), 404
invariant_form() (sage.groups.matrix_gps.unitary.UnitaryMatrixGroup_generic method), 401
is_abelian() (sage.groups.libgap_mixin.GroupMixinLibGAP method), 392
is_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 3
is_abelian() (sage.groups.perm_gps.permgroup_named.CyclicPermutationGroup method), 294
is_abelian() (sage.groups.perm_gps.permgroup_named.DiCyclicGroup method), 28
is_abelian() (sage.groups.perm_gps.permgroup_named.DiCyclicGroup method), 163
is_abelian() (sage.groups.perm_gps.permgroup_named.CyclicPermutationGroup method), 294
is_abelian() (sage.groups.perm_gps.permgroup_named.CyclicPermutationGroup method), 179
is_abelian() (sage.groups.perm_gps.permgroup_named.CyclicPermutationGroup method), 3
is_abelian() (sage.groups.perm_gps.permgroup_named.DiCyclicGroup method), 142
is_abelian() (sage.groups.perm_gps.permgroup_named.DiCyclicGroup method), 100
is_abelian() (sage.groups.perm_gps.permgroup_named.DiCyclicGroup method), 3
is_abelian() (sage.groups.perm_gps.permgroup_named.DiCyclicGroup method), 179
is_abelian() (sage.groups.perm_gps.permgroup_named.DiCyclicGroup method), 264
is_cyclic() (sage.groups.abelian_gps.dual_abelian_group.DualAbelianGroup_class method), 94
is_cyclic() (sage.groups.abelian_gps.dual_abelian_group.DualAbelianGroup_class method), 152
is_cyclic() (sage.groups.abelian_gps.dual_abelian_group.DualAbelianGroup_class method), 380
is_cyclic() (sage.groups.abelian_gps.dual_abelian_group.DualAbelianGroup_class method), 14
is_cyclic() (sage.groups.abelian_gps.dual_abelian_group.DualAbelianGroup_class method), 416
is_cyclic() (sage.groups.abelian_gps.dual_abelian_group.DualAbelianGroup_class method), 264
is_cyclic() (sage.groups.abelian_gps.dual_abelian_group.DualAbelianGroup_class method), 416
is_cyclic() (sage.groups.abelian_gps.dual_abelian_group.DualAbelianGroup_class method), 380
is_commutative() (sage.groups.abelian_gps.abelian_group.AbelianGroup_class method), 207
is_commutative() (sage.groups.abelian_gps.abelian_group.AbelianGroup_class method), 264
is_commutative() (sage.groups.abelian_gps.abelian_group.AbelianGroup_class method), 3
is_commutative() (sage.groups.abelian_gps.abelian_group.AbelianGroup_class method), 207
is_commutative() (sage.groups.abelian_gps.abelian_group.AbelianGroup_class method), 213
is_commutative() (sage.groups.abelian_gps.abelian_group.AbelianGroup_class method), 217
is_commutative() (sage.groups.abelian_gps.abelian_group.AbelianGroup_class method), 215
is_commutative() (sage.groups.abelian_gps.abelian_group.AbelianGroup_class method), 264
is_commutative() (sage.groups.abelian_gps.abelian_group.AbelianGroup_class method), 324
is_conjugated() (sage.groups.braid.Braid method), 94
is_conjugated() (sage.groups.braid.Braid method), 3
is_conjugated() (sage.groups.braid.Braid method), 152
is_conjugated() (sage.groups.braid.Braid method), 3
is_conjugated() (sage.groups.braid.Braid method), 152
is_conjugated() (sage.groups.braid.Braid method), 94
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 208
is_elementary_abelian() (sage/groups/semimonomial_transformations/semimonomial_transformation/ SemimonomialTransformation method), 152
is_elementary_abelian() (sage/groups/semimonomial_transformations/semimonomial_transformation/ SemimonomialTransformation method), 3
is_elementary_abelian() (sage/groups/semimonomial_transformations/semimonomial_transformation/ SemimonomialTransformation method), 152
is_elementary_abelian() (sage/groups/semimonomial_transformations/semimonomial_transformation/ SemimonomialTransformation method), 3
is_elementary_abelian() (sage/groups/semimonomial_transformations/semimonomial_transformation/ SemimonomialTransformation method), 152
is_elementary_abelian() (sage/groups/semimonomial_transformations/semimonomial_transformation/ SemimonomialTransformation method), 3
log() (sage.groups.lie_gps.nilpotent_lie_group.NilpotentLieGroup.method), 427
lower_central_series() (sage.groups.perm_gps.permgroup.PermutationGroup_generic.method), 271
M
major_index() (sage.groups.perm_gps.permgroup_named.SymmetricGroup.method), 315
make_confluent() (sage.groups.finitely_presented.RewritingSystem.method), 75
make_permgroup_element() (in module sage.groups.perm_gps.permgroup_element), 328
make_permgroup_element_v2() (in module sage.groups.perm_gps.permgroup_element), 328
mapping_class_action() (sage.groups.braid.BraidGroup_class.method), 105
MathieuGroup (class in sage.groups.perm_gps.permgroup_named), 299
matrix() (sage.groups.abelian_gps.abelian_aut.AbelianGroupAutomorphism.method), 197
matrix() (sage.groups.abelian_gps.abelian_group.AbelianGroupElement.method), 171
matrix() (sage.groups.abelian_gps.abelian_group_element.AbelianGroupElement_element_gmp_int.method), 213
matrix() (sage.groups.abelian_gps.abelian_group_gap.AbelianGroupElement.method), 187
matrix() (sage.groups.abelian_gps.dual_abelian_group.DualAbelianGroupElement.method), 213
matrix() (sage.groups.abelian_gps.dual_abelian_group_element.DualAbelianGroupElement_element_gmp_int.method), 205
matrix() (sage.groups.abelian_gps.abelian_group.morphism.AbelianGroupMorphismElement.method), 219
matrix_degree() (sage.groups.perm_gps.permgroup_named.PermutationGroup_plg.method), 304
matrix_space() (in module sage.groups.perm_gps.permgroup_element.PermutationGroupElement), 325
MatrixGroup() (in module sage.groups.matrix_gps.finitely_generated), 371
MatrixGroup_base (class in sage.groups.matrix_gps.matrix_group), 349
MatrixGroup_gap (class in sage.groups.matrix_gps.matrix_group), 351
MatrixGroup_generic (class in sage.groups.matrix_gps.matrix_group), 353
MatrixGroupElement_gap (class in sage.groups.matrix_gps.group_element), 355
MatrixGroupElement_generic (class in sage.groups.matrix_gps.group_element), 358
MatrixStruct (class in sage.groups.perm_gps.partn_ref.refinement_matrices), 442
merge_points() (in module sage.groups.generic), 45
minimal_generating_set() (sage.groups.perm_gps.permgroup.PermutationGroup_generic.method), 271
mirror_image() (sage.groups.braid.Braid.method), 97
mirror_involution() (sage.groups.braid.BraidGroup_class.method), 106
module
sage.groups.abelian_gps.abelian_aut, 195
sage.groups.abelian_gps.abelian_group, 171
sage.groups.abelian_gps.abelian_group_element, 211
sage.groups.abelian_gps.abelian_group_gap, 187
sage.groups.abelian_gps.abelian_group_morphism, 215
sage.groups.abelian_gps.dual_abelian_group_element, 213
sage.groups.abelian_gps.dual_abelian_group_element_element_gmp_int, 219
sage.groups.abelian_gps.element_base, 208
sage.groups.abelian_gps.values, 200
sage.groups.additive_abelian.additive_abelian_group, 217
sage.groups.additive_abelian.additive_abelian_group_element, 221
sage.groups.additive_abelian.additive_abelian_group_element_element_gmp_int, 223
sage.groups.additive_group, 223
sage.groups.additive_group.element_base, 225
sage.groups.additive_group.element_gmp_int, 227
sage.groups.additive_group.morphism, 229
sage.groups.additive_group.morphism_element_gmp_int, 231
sage.groups.additive_group.morphism_element_pyobject, 233
sage.groups.additive_group.morphism_pyobject, 235
sage.groups.additive_group.pyobject, 237
sage.groups.conjugacy_classes, 167
sage.groups.cubic_braid, 111
sage.groups.finitely_presented, 59
sage.groups.finitely_presented_named, 79
sage.groups.free_group, 51
sage.groups.generic, 37
sage.groups.group, 3
sage.groups.group_exp, 135
sage.groups.group_semidirect_product, 139
sage.groups.groups_catalog, 1
sage.groups.indexed_free_group, 125
sage.groups.libgap_group, 21
sage.groups.libgap_mixin, 23
sage.groups.libgap_morphism, 7

Index 469

non_fixed_points() (sage.groups.perm_gps.permgroup.PermutationGroup_generic.

non_fixed_points(), 272

norm() (sage.groups.class_function.ClassFunction_gap.

norm(), 158

norm() (sage.groups.class_function.ClassFunction_libgap.

norm(), 164

normal_subgroups() (sage.groups.perm_gps.permgroup.

normal_subgroups(), 273

normalize_args_e() (in module

normalize_args_e(), 394

normalize_args_invariant_form() (in module

normalize_args_invariant_form(), 446

normalize_args_vectorspace() (in module

normalize_args_vectorspace(), 447

normalize_square_matrices() (in module

normalize_square_matrices(), 372

normalizer() (sage.groups.libgap_wrapper.ElementLibGAP.

normalizer(), 15

normalizer() (sage.groups.perm_gps.permgroup.

normalizer(), 273

n th_roots() (sage.groups.libgap_wrapper.ElementLibGA.

n th_roots(), 15

number_of_subgroups() (sage.groups.abelian_gps.abelian_group.

number_of_subgroups(), 181

one() (sage.groups.group_exp.GroupExp_Class.

one(), 137

one() (sage.groups.group_semidirect_product.

one(), 141

one() (sage.groups.indexed_free_group.

one(), 127

one() (sage.groups.libgap_wrapper.

one(), 19

one() (sage.groups. lie_gps.nilpotent_lie_group.

one(), 427

one() (sage.groups.perm_gps.permgroup.

one(), 274

one() (sage.groups.raag.

one(), 133

one_basis() (sage.groups.raag.

one_basis(), 130

one_element() (sage.groups.raag.

one_element(), 133

OP_pose() (in module

OP_pose(), 433

opposite_semidirect_product() (sage.groups.

opposite_semidirect_product(), 141

orbit() (sage.groups.perm_gps.permgroup.

orbit(), 274

orbit() (sage.groups.perm_gps.permgroup_element.

orbit(), 325

orbit() (sage.groups.libgap_wrapper.

orbit(), 436

orbits() (sage.groups.perm_gps.permgroup.

orbits(), 275

order() (sage.groups.abelian_gps.

order(), 181

order() (sage.groups.abelian_gps.dual_abelian_group.

order(), 188

order() (sage.groups.abelian_gps.

order(), 272

order() (sage.groups.braid.

order(), 106

order() (sage.groups.cubic_braid.

order(), 123

order() (sage.groups.finitely_presented.

order(), 66

order() (sage.groups.

order(), 4

order() (sage.groups.

order(), 127

order() (sage.groups.libgap_mixin.

order(), 31

order() (sage.groups.libgap_wrapper.

order(), 16

order() (sage.groups.matrix_gps.

order(), 374

order() (sage.groups.

order(), 381

order() (sage.groups.

order(), 407

order() (sage.groups.

order(), 36

order() (sage.groups.

order(), 276

order() (sage.groups.

order(), 149

order_from_bounds() (in module

order_from_bounds(), 47

order_from_multiple() (in module

order_from_multiple(), 48

OrthogonalMatrixGroup_gap (class in

OrthogonalMatrixGroup_gap(), 389

OrthogonalMatrixGroup_generic (class in
Index

sage.groups.matrix_gps.orthogonal), 391

P

ParentLibGAP (class in sage.groups.libgap_wrapper), 16

PariGroup (class in sage.groups.pari_group), 35

parse() (sage.groups.perm_gps.cubegroup.CubeGroup method), 336

partition() (sage.groups.perm_gps.symgp_conjugacy_class.SymmetricGroupConjugacyClassMixin method), 345

permutation() (sage.groups.braid.Braid method), 97

permutation_group() (sage.groups.abelian_gps.abelian_group.AbelianGroup_class method), 182

permutation_group() (sage.groups.additive_abelian.additive_abelian_group.AdditiveAbelianGroup_fixed_gens method), 220

permutation_group() (sage.groups.pari_group.PariGroup method), 36

PermutationGroup() (in module sage.groups.perm_gps.permgroup), 240

PermutationGroup_generic (class in sage.groups.perm_gps.permgroup), 241

PermutationGroup_plg (class in sage.groups.perm_gps.permgroup_named), 304

PermutationGroup_pug (class in sage.groups.perm_gps.permgroup_named), 304

PermutationGroup_subgroup (class in sage.groups.perm_gps.permgroup), 286

PermutationGroup_symalt (class in sage.groups.perm_gps.permgroup_named), 304

PermutationGroup_unique (class in sage.groups.perm_gps.permgroup_named), 305

PermutationGroupElement (class in sage.groups.perm_gps.permgroup_element), 320

PermutationGroupElement() (in module sage.groups.perm_gps.constructor), 233

PermutationGroupMorphism (class in sage.groups.perm_gps.permgroup_morphism), 329

PermutationGroupMorphism_from_gap (class in sage.groups.perm_gps.permgroup_morphism), 330

PermutationGroupMorphism_id (class in sage.groups.perm_gps.permgroup_morphism), 330

PermutationGroupMorphism_im_gens (class in sage.groups.perm_gps.permgroup_morphism), 330

PermutationsConjugacyClass (class in sage.groups.perm_gps.symgp_conjugacy_class), 345

PGL (class in sage.groups.perm_gps.permgroup_named), 300

PGU (class in sage.groups.perm_gps.permgroup_named), 300

plot() (sage.groups.braid.Braid method), 98

plot() (sage.groups.perm_gps.cubegroup.RubiksCube method), 340

plot3d() (sage.groups.braid.Braid method), 98

plot3d() (sage.groups.perm_gps.cubegroup.RubiksCube method), 340

plot3d_cubie() (sage.groups.perm_gps.cubegroup.CubeGroup method), 337

plot3d_cube() (sage.groups.perm_gps.cubegroup.CubeGroup method), 343

plot_cube() (sage.groups.perm_gps.cubegroup.CubeGroup method), 337

poincare_series() (sage.groups.perm_gps.permgroup.PermutationGroup method), 276

polygon_plot3d() (in module sage.groups.perm_gps.cubegroup), 344

positive_roots() (sage.groups.matrix_gps.coxeter_group.CoxeterMatrixGroup method), 381

preimage() (sage.groups.libgap_morphism.GroupMorphism_libgap method), 10

PrimitiveGroup (class in sage.groups.perm_gps.permgroup_named), 305

PrimitiveGroups() (in module sage.groups.perm_gps.permgroup_named), 306

PrimitiveGroupsAll (class in sage.groups.perm_gps.permgroup_named), 306

PrimitiveGroupsOfDegree (class in sage.groups.perm_gps.permgroup_named), 307

product() (sage.groups.group_exp.GroupExp_Class method), 137

product() (sage.groups.group_semidirect_product.GroupSemidirectProduct method), 142

PS_reps() (in module sage.groups.perm_gps.partn_ref.data_structures), 433

PSL (class in sage.groups.perm_gps.permgroup_named), 301

PSp (class in sage.groups.perm_gps.permgroup_named), 303

PSP (in module sage.groups.perm_gps.permgroup_named), 303

PSU (in module sage.groups.perm_gps.permgroup_named), 303

PGL (class in sage.groups.perm_gps.permgroup_named), 300

PGU (class in sage.groups.perm_gps.permgroup_named), 300

plot() (sage.groups.braid.Braid method), 98

plot() (sage.groups.perm_gps.cubegroup.RubiksCube method), 340

plot3d() (sage.groups.braid.Braid method), 98

plot3d() (sage.groups.perm_gps.cubegroup.RubiksCube method), 340

plot3d_cubie() (sage.groups.perm_gps.cubegroup.CubeGroup method), 337

plot3d_cube() (sage.groups.perm_gps.cubegroup.CubeGroup method), 343

plot_cube() (sage.groups.perm_gps.cubegroup.CubeGroup method), 337

poincare_series() (sage.groups.perm_gps.permgroup.PermutationGroup method), 276

polygon_plot3d() (in module sage.groups.perm_gps.cubegroup), 344

positive_roots() (sage.groups.matrix_gps.coxeter_group.CoxeterMatrixGroup method), 381

preimage() (sage.groups.libgap_morphism.GroupMorphism_libgap method), 10

PrimitiveGroup (class in sage.groups.perm_gps.permgroup_named), 305

PrimitiveGroups() (in module sage.groups.perm_gps.permgroup_named), 306

PrimitiveGroupsAll (class in sage.groups.perm_gps.permgroup_named), 306

PrimitiveGroupsOfDegree (class in sage.groups.perm_gps.permgroup_named), 307

product() (sage.groups.group_exp.GroupExp_Class method), 137

product() (sage.groups.group_semidirect_product.GroupSemidirectProduct method), 142

PS_reps() (in module sage.groups.perm_gps.partn_ref.data_structures), 433

PSL (class in sage.groups.perm_gps.permgroup_named), 301

PSp (class in sage.groups.perm_gps.permgroup_named), 303

PSP (in module sage.groups.perm_gps.permgroup_named), 303

PSU (in module sage.groups.perm_gps.permgroup_named), 303
pushforward() (sage.groups.libgap_morphism.GroupMorphism_libgap method), 11

QuaternionGroup (class in sage.groups.perm_gps.permgroup_named), 308
QuaternionMatrixGroupGF3() (in module sage.groups.matrix_gps.finitely_generated), 372
QuaternionPresentation() (in module sage.groups.finitely_presented_named), 83
quotient() (sage.groups.finitely_presented.FinitelyPresentedGroup method), 194
quotient() (sage.groups.abelian_gps.abelian_group_quotient method), 190
quotient() (sage.groups.perm_gps.permgroup_group_quotient method), 277

R
R() (sage.groups.perm_gps.cubegroup.CubeGroup method), 333
ramification_module_decomposition_hurwitz_curve() (sage.groups.perm_gps.permgroup_named.PSL method), 302
ramification_module_decomposition_modular_curve() (sage.groups.perm_gps.permgroup_named.PSL method), 302
random_element() (sage.groups.abelian_gps.abelian_group_AbelianGroup method), 182
random_element() (sage.groups.abelian_gps.dual_abelian_group_duality_element method), 208
random_element() (sage.groups.finitely_presented.RewritingSystem method), 410
random_element() (sage.groups.finitely_presented.FinitelyPresentedGroup method), 414
random_element() (sage.groups.matrix_gps.coxeter_group.CoxeterMatrixGroup method), 277
random_tests() (in module sage.groups.perm_gps.partn_ref.refinement_graphs), 437
random_tests() (in module sage.groups.perm_gps.partn_ref.refinement_matrices), 443
rank() (sage.groups.free_group.FreeGroup_class method), 56
rank() (sage.groups.indexed_free_group.IndexedGroup method), 128
real() (sage.groups.misc_gps.imaginary_groups.ImaginaryElement method), 230
reduce() (sage.groups.finitely_presented.RewritingSystem method), 76
reduced_word() (sage.groups.braid.RightQuantumWord method), 108
reflection() (sage.groups.affine_gps.affine_group.AffineGroup method), 108
reflections() (sage.groups.matrix_gps.coxeter_group.CoxeterMatrixGroup method), 382
reflections() (sage.groups.perm_gps.permgroup_named.SymmetricGroup method), 315
relations() (sage.groups.abelian_gps.abelian_group_AbelianGroup method), 190
relations() (sage.groups.finitely_presented.FinitelyPresentedGroup method), 66
repr2d() (sage.groups.perm_gps.cubegroup.CubeGroup method), 337

representative() (sage.groups.conjugacy_classes.ConjugacyClass method), 168
representative_action() (sage.groups.perm_gps.permgroup_permutationgroup.PermutationGroup_generic method), 277
restrict() (sage.groups.class_function.ClassFunction_gap method), 159
restrict() (sage.groups.class_function.ClassFunction_libgap method), 159
retract() (sage.groups.abelian_gps.abelian_group_AbelianGroup method), 191
reverse() (sage.groups.braid.Braid method), 99
rewriting_system() (sage.groups.finitely_presented.FinitelyPresentedGroup method), 67

RewritingSystem (class in sage.groups.finitely_presented), 72
reynolds_operator() (sage.groups.matrix_gps.finitely_generated.FinitelyGeneratedMatrixGroup_gap method), 367

RightInvariantExtension() (sage.groups.libgap_morphism.ReynoldsOperator_method), 427
right_invariant_extension() (sage.groups.libgap_mixin.GroupMorphismLieGroup method), 428
right_invariant_frame() (sage.groups.libgap_morphism.ReynoldsOperator_method), 428

right_normal_form() (sage.groups.braid.Braid method), 99
right_translation() (sage.groups.libgap_morphism.ReynoldsOperator_method), 428

RightAngledArtinGroup (class in sage.groups.raag), 130
RightAngledArtinGroup.Element (class in sage.groups.raag), 132
RightQuantumWord (class in sage.groups.braid), 107
rigidity() (sage.groups.braid.Braid method), 99

rivf() (sage.groups.libgap_morphism.ReynoldsOperator_method), 429

RootOfUnity (class in sage.groups.misc_gps.argument_groups), 226
roots() (sage.groups.matrix_gps.coxeter_group.CoxeterMatrixGroup method), 382
RootsOfUnityGroup (class in sage.groups.misc_gps.argument_groups), 227
rotation_list() (in module sage.groups.perm_gps.cubegroup), 344
RubiksCube (class in sage.groups.perm_gps.cubegroup), 339
rules() (sage.groups.finitely_presented.RewritingSystem method), 76
run() (sage.groups.perm_gps.partn_ref.refinement_matrices.MatrixStruct method), 443
sage.groups.abelian_gps.abelian_aut module, 195
sage.groups.abelian_gps.abelian_gap module, 211
sage.groups.abelian_gps.abelian_group module, 171
sage.groups.abelian_gps.abelian_group_element module, 215
sage.groups.abelian_gps.abelian_group_gap module, 211
sage.groups.abelian_gps.abelian_group_morphism module, 211
sage.groups.abelian_gps.dual_abelian_group module, 205
sage.groups.abelian_gps.dual_abelian_group_element module, 213
sage.groups.abelian_gps.element_base module, 208
sage.groups.abelian_gps.values module, 200
sage.groups.additive_abelian.additive_abelian_group module, 217
sage.groups.additive_abelian.additive_abelian_group_element module, 221
sage.groups.affine_gps.affine_group module, 407
sage.groups.affine_gps.euclidean_group module, 412
sage.groups.affine_gps.group_element module, 415
sage.groups.braid module, 85
sage.groups.class_function module, 155
sage.groups.conjugacy_classes module, 167
sage.groups.cubic_braid module, 111
sage.groups.finitely_presented module, 59
sage.groups.finitely_presented_named module, 79
sage.groups.free_group module, 51
sage.groups.generic module, 37
sage.groups.group module, 3
sage.groups.group_exp module, 135
sage.groups.group_semidirect_product module, 139
sage.groups.indexed_free_group module, 125
sage.groups.libgap_group module, 21
sage.groups.libgap_mixin module, 23
sage.groups.libgap_morphism module, 7
sage.groups.libgap_wrapper module, 13
sage.groups.lie_gps.nilpotent_lie_group module, 419
sage.groups.matrix_gps.binary_dihedral module, 374
sage.groups.matrix_gps.coxeter_group module, 349
sage.groups.matrix_gps.finitely_generated module, 360
sage.groups.matrix_gps.group_element module, 354
sage.groups.matrix_gps.heisenberg module, 406
sage.groups.matrix_gps.homset module, 373
sage.groups.matrix_gps.isometries module, 395
sage.groups.matrix_gps.linear module, 383
sage.groups.matrix_gps.matrix_group module, 349
sage.groups.matrix_gps.morphism module, 373
sage.groups.matrix_gps.named_group module, 445
sage.groups.matrix_gps.orthogonal module, 386
simple_reflection()
(sage.groups.perm_gps.permgroup_named.ComplexReflectionGroup(sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 293)

structure_description()
(in module sage.groups.finitely_presented.FinitelyPresentedGroup, 49)

structure_description()
(sage.groups.finitely_presented.FinitelyPresentedGroup(sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 282)

simplified()
(sage.groups.finitely_presented.FinitelyPresentedGroup(sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 69)

sliding_circuits()
(sage.groups.braid.Braid method), 99

smallest_moved_point()
(sage.groups.perm_gps.permgroup.PermutationGroup_gap method), 279

SO()
(in module sage.groups.matrix_gps.orthogonal), 393

socle()
(sage.groups.perm_gps.permgroup.PermutationGroup(sage.groups.matrix_gps.matrix_group.MatrixGroup_gap method), 280)

solvable_radical()
(sage.groups.perm_gps.permgroup.PermutationGroup(sage.groups.matrix_gps.matrix_group.MatrixGroup_base method), 280)

solve()
(sage.groups.perm_gps.cubegroup.CubeGroup method), 338

solve()
(sage.groups.perm_gps.cubegroup.CubeGroup method), 341

solve()
(sage.groups.perm_gps.cubegroup.CubeGroup method), 309

some_elements()
(sage.groups.affine_gps.affine_group.AffineGroup_class method), 411

some_elements()
(sage.groups.braid.BraidGroup_class method), 106

Sp()
(in module sage.groups.matrix_gps.symplectic), 398

SuzukiSporadicGroup
(class in sage.groups.perm_gps.permgroup_named), 310

SuzukiSporadicGroup
(class in sage.groups.perm_gps.permgroup_named), 311

stabilizer()
(sage.groups.perm_gps.permgroup.PermutationGroup(sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 280)

standardize_generator()
(in module sage.groups.perm_gps.constructor), 234

step()
(sage.groups.lie_gps.nilpotent_lie_group.NilpotentLieGroup(sage.groups.perm_gps.permgroup.PermutationGroup_generic method), 430)

strands()
(sage.groups.braid.Braid method), 100

symmetric_power()
(sage.groups.class_function.ClassFunction_gap method), 285

symmetric_power()
(sage.groups.class_function.ClassFunction_libgap method), 164

SymmetricGroup
(class in sage.groups.perm_gps.permgroup_named), 311

SymmetricGroupConjugacyClass
(class in
Index
values() (sage.groups.class_function.ClassFunction_libgap method), 165
values_embedding() (sage.groups.abelian_gps.values.AbelianGroupWithValues_class method), 204
values_group() (sage.groups.abelian_gps.values.AbelianGroupWithValues_class method), 204
vector_space() (sage.groups.affine_gps.affine_group.AffineGroup method), 412

W
word_problem() (in module sage.groups.abelian_gps.abelian_group), 185
word_problem() (sage.groups.abelian_gps.abelian_group_element.AbelianGroupElement method), 212
word_problem() (sage.groups.abelian_gps.dual_abelian_group_element.DualAbelianGroupElement method), 214
word_problem() (sage.groups.matrix_gps.group_element.MatrixGroupElement_gap method), 357
word_problem() (sage.groups.perm_gps.permgroup_element.PermutationGroupElement method), 326
wrap_FpGroup() (in module sage.groups.finitely_presented), 76
wrap_FreeGroup() (in module sage.groups.free_group), 57

X
xproj() (in module sage.groups.perm_gps.cubegroup), 344

Y
young_subgroup() (sage.groups.perm_gps.permgroup_named.SymmetricGroup method), 315
yproj() (in module sage.groups.perm_gps.cubegroup), 344