## CONTENTS

1 ECL 3
   1.1 Library interface to Embeddable Common Lisp (ECL) 3

2 eclib 11
   2.1 Sage interface to Cremona’s eclib library (also known as mwrank) 11
   2.2 Cython interface to Cremona’s eclib library (also known as mwrank) 26
   2.3 Cremona matrices 28
   2.4 Modular symbols using eclib newforms 29
   2.5 Cremona modular symbols 31
   2.6 Cremona modular symbols 34

3 FLINT 37
   3.1 Flint imports 37
   3.2 FLINT fmpz_poly class wrapper 37
   3.3 FLINT Arithmetic Functions 39

4 GMP-ECM 43
   4.1 The Elliptic Curve Method for Integer Factorization (ECM) 43

5 GSL 45
   5.1 GSL arrays 45

6 lcalc 47
   6.1 Rubinstein’s lcalc library 47

7 libSingular 55
   7.1 libSingular: Functions 55
   7.2 libSingular: Function Factory 63
   7.3 libSingular: Conversion Routines and Initialisation 63
   7.4 Wrapper for Singular’s Polynomial Arithmetic 64
   7.5 libSingular: Options 64
   7.6 Wrapper for Singular’s Rings 69
   7.7 Singular’s Groebner Strategy Objects 70

8 GAP 75
   8.1 Context Managers for LibGAP 75
   8.2 Common global functions defined by GAP 76
   8.3 Long tests for GAP 76
   8.4 Utility functions for GAP 76
   8.5 Library Interface to GAP 77
   8.6 Short tests for GAP 83
An underlying philosophy in the development of Sage is that it should provide unified library-level access to some of the best GPL’d C/C++ libraries. Sage provides access to many libraries which are included with Sage.

The interfaces are implemented via shared libraries and data is moved between systems purely in memory. In particular, there is no interprocess interpreter parsing (e.g., `pexpect`), since everything is linked together and run as a single process. This is much more robust and efficient than using `pexpect`.

Each of these interfaces is used by other parts of Sage. For example, eclib is used by the elliptic curves module to compute ranks of elliptic curves and PARI is used for computation of class groups. It is thus probably not necessary for a casual user of Sage to be aware of the modules described in this chapter.
1.1 Library interface to Embeddable Common Lisp (ECL)

class sage.libs.ecl.EclListIterator
Bases: object

Iterator object for an ECL list

This class is used to implement the iterator protocol for EclObject. Do not instantiate this class directly but use the iterator method on an EclObject instead. It is an error if the EclObject is not a list.

EXAMPLES:

```python
sage: from sage.libs.ecl import *
sage: I=EclListIterator(EclObject("(1 2 3)"))
sage: type(I)
<type 'sage.libs.ecl.EclListIterator'>
sage: [i for i in I]
[<ECL: 1>, <ECL: 2>, <ECL: 3>]
sage: [i for i in EclObject("(1 2 3)")]
[<ECL: 1>, <ECL: 2>, <ECL: 3>]
sage: EclListIterator(EclObject("1"))
Traceback (most recent call last):
... TypeError: ECL object is not iterable
```

next ()

x.next() -> the next value, or raise StopIteration

class sage.libs.ecl.EclObject
Bases: object

Python wrapper of ECL objects

The EclObject forms a wrapper around ECL objects. The wrapper ensures that the data structure pointed to is protected from garbage collection in ECL by installing a pointer to it from a global data structure within the scope of the ECL garbage collector. This pointer is destroyed upon destruction of the EclObject. EclObject() takes a Python object and tries to find a representation of it in Lisp.

EXAMPLES:

Python lists get mapped to LISP lists. None and Boolean values to appropriate values in LISP:

```python
sage: from sage.libs.ecl import *
sage: EclObject([None,true,false])
<ECL: (NIL T NIL)>
```
Numerical values are translated to the appropriate type in LISP:

```
sage: EclObject(1)
<ECL: 1>
sage: EclObject(10**40)
<ECL: 10000000000000000000000000000000000000000>
```

Floats in Python are IEEE double, which LISP has as well. However, the printing of floating point types in LISP depends on settings:

```
sage: a = EclObject(float(10^40))
sage: ecl_eval("(setf *read-default-float-format* 'single-float)")
<ECL: SINGLE-FLOAT>
sage: a
<ECL: 1.d40>
sage: ecl_eval("(setf *read-default-float-format* 'double-float)")
<ECL: DOUBLE-FLOAT>
sage: a
<ECL: 1.e40>
```

Tuples are translated to dotted lists:

```
sage: EclObject((false, true))
<ECL: (NIL . T)>
```

Strings are fed to the reader, so a string normally results in a symbol:

```
sage: EclObject("Symbol")
<ECL: SYMBOL>
```

But with proper quotation one can construct a lisp string object too:

```
sage: EclObject("Symbol'")
<ECL: "Symbol"> 
```

EclObjects translate to themselves, so one can mix:

```
sage: EclObject([1,2,EclObject([3])])
<ECL: (1 2 (3))>
```

Calling an EclObject translates into the appropriate LISP `apply`, where the argument is transformed into an EclObject itself, so one can flexibly apply LISP functions:

```
sage: car=EclObject("car")
sage: cdr=EclObject("cdr")
sage: car(cdr([1,2,3]))
<ECL: 2>
```

and even construct and evaluate arbitrary S-expressions:

```
sage: eval=EclObject("eval")
sage: quote=EclObject("quote")
sage: eval([car, [cdr, [quote,[1,2,3]]]])
<ECL: 2>
```

```
atomp()
Return True if self is atomic, False otherwise.
```

EXAMPLES:
caar()  
Return the caar of self

EXAMPLES:

```python
sage: from sage.libs.ecl import *
sage: L=EclObject([[1,2],[3,4]])
sage: L.car()
<ECL: (1 2)>
sage: L.cdr()
<ECL: (3 4)>
sage: L.caar()
<ECL: 1>
sage: L.cadr()
<ECL: (3 4)>
sage: L.cdar()
<ECL: (2)>
sage: L.cddr()
<ECL: NIL>
```

cadr()  
Return the cadr of self

EXAMPLES:

```python
sage: from sage.libs.ecl import *
sage: L=EclObject([[1,2],[3,4]])
sage: L.car()
<ECL: (1 2)>
sage: L.cdr()
<ECL: (3 4)>
sage: L.caar()
<ECL: 1>
sage: L.cadr()
<ECL: (3 4)>
sage: L.cdar()
<ECL: (2)>
sage: L.cddr()
<ECL: NIL>
```

car()  
Return the car of self

EXAMPLES:

```python
sage: from sage.libs.ecl import *
sage: L=EclObject([[1,2],[3,4]])
sage: L.car()
<ECL: (1 2)>
sage: L.cdr()
<ECL: (3 4)>
sage: L.caar()
```

(continues on next page)
cdar ()
Return the cdar of self

EXAMPLES:

```python
sage: from sage.libs.ecl import *
sage: L=EclObject([[1,2],[3,4]])
sage: L.car()
<ECL: (1 2)>
sage: L.cdr()
<ECL: ((3 4))>
sage: L.caar()
<ECL: 1>
sage: L.cadr()
<ECL: (3 4)>
sage: L.cdar()
<ECL: (2)>
sage: L.cddr()
<ECL: NIL>
```

cddr ()
Return the cddr of self

EXAMPLES:

```python
sage: from sage.libs.ecl import *
sage: L=EclObject([[1,2],[3,4]])
sage: L.car()
<ECL: (1 2)>
sage: L.cdr()
<ECL: ((3 4))>
sage: L.caar()
<ECL: 1>
sage: L.cadr()
<ECL: (3 4)>
sage: L.cdar()
<ECL: (2)>
sage: L.cddr()
<ECL: NIL>
```

cdr ()
Return the cdr of self

EXAMPLES:

```python
sage: from sage.libs.ecl import *
sage: L=EclObject([[1,2],[3,4]])
sage: L.car()
<ECL: (1 2)>
```

(continues on next page)
characterp()
Return True if self is a character, False otherwise

Strings are not characters

EXAMPLES:

```python
sage: from sage.libs.ecl import *
sage: EclObject('"a"').characterp()
False
```

cons (d)
apply cons to self and argument and return the result.

EXAMPLES:

```python
sage: from sage.libs.ecl import *
sage: a=EclObject(1)
sage: b=EclObject(2)
sage: a.cons(b)
<ECL: (1 . 2)>
```

consp()
Return True if self is a cons, False otherwise. NIL is not a cons.

EXAMPLES:

```python
sage: from sage.libs.ecl import *
sage: EclObject([]).consp()
False
sage: EclObject([[]]).consp()
True
```

eval()
Evaluate object as an S-Expression

EXAMPLES:

```python
sage: from sage.libs.ecl import *
sage: S=EclObject("(+ 1 2)")
sage: S
<ECL: (+ 1 2)>
sage: S.eval()
<ECL: 3>
```

fixnump()
Return True if self is a fixnum, False otherwise

1.1. Library interface to Embeddable Common Lisp (ECL)
EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: EclObject(2**3).fixnump()
True
sage: EclObject(2**200).fixnump()
False
```

`listp()`
Return True if self is a list, False otherwise. NIL is a list.

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: EclObject([]).listp()
True
sage: EclObject([[]]).listp()
True
```

`nullp()`
Return True if self is NIL, False otherwise

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: EclObject([]).nullp()
True
sage: EclObject([[]]).nullp()
False
```

`python()`
Convert an EclObject to a python object.

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: L=EclObject([1,2,('three','"four"')])
sage: L.python()
[1, 2, ('THREE', '"four"')]
```

`rplaca(d)`
Destructively replace car(self) with d.

EXAMPLES:

```
sage: from sage.libs.ecl import *
sage: L=EclObject((1,2))
sage: L
<ECL: (1 . 2)>
sage: a=EclObject(3)
sage: L.rplaca(a)
sage: L
<ECL: (3 . 2)>
```

`rplacd(d)`
Destructively replace cdr(self) with d.

EXAMPLES:
sage: from sage.libs.ecl import *
sage: L=EclObject((1,2))
sage: L
<ECL: (1 . 2)>
sage: a=EclObject(3)
sage: L.rplacd(a)
sage: L
<ECL: (1 . 3)>

symbolp()
Return True if self is a symbol, False otherwise.

EXAMPLES:

sage: from sage.libs.ecl import *
sage: EclObject([]).symbolp()
True
sage: EclObject([[]]).symbolp()
False

sage.libs.ecl.ecl_eval(s)
Read and evaluate string in Lisp and return the result

EXAMPLES:

sage: from sage.libs.ecl import *
sage: ecl_eval("(defun fibo (n)(cond((= n 0) 0)((= n 1) 1)(T (+ (fibo (- n 1))
˓→(fibo (- n 2))))))")
<ECL: FIBO>
sage: ecl_eval("(mapcar 'fibo '(1 2 3 4 5 6 7))")
<ECL: (1 1 2 3 4 5 6 7)>

sage.libs.ecl.init_ecl()
Internal function to initialize ecl. Do not call.

This function initializes the ECL library for use within Python. This routine should only be called once and importing the ecl library interface already does that, so do not call this yourself.

EXAMPLES:

sage: from sage.libs.ecl import *

At this point, init_ecl() has run. Explicitly executing it gives an error:

sage: init_ecl()  
Traceback (most recent call last):
  ... 
RuntimeError: ECL is already initialized

sage.libs.ecl.print_objects()
Print GC-protection list

Diagnostic function. ECL objects that are bound to Python objects need to be protected from being garbage collected. We do this by including them in a doubly linked list bound to the global ECL symbol SAGE-LIST-OF-OBJECTS. Only non-immediate values get included, so small integers do not get linked in. This routine prints the values currently stored.

EXAMPLES:
```python
sage: from sage.libs.ecl import *
sage: a=EclObject("hello")
sage: b=EclObject(10)
sage: c=EclObject("world")
sage: print_objects()  #random because previous test runs can have left objects
NIL
WORLD
HELLO
```

sage.libs.ecl.shutdown_ecl()

Shut down ecl. Do not call.

Given the way that ECL is used from python, it is very difficult to ensure that no ECL objects exist at a particular time. Hence, destroying ECL is a risky proposition.

EXAMPLES:

```python
sage: from sage.libs.ecl import *
sage: shutdown_ecl()
```

sage.libs.ecl.test_ecl_options()

Print an overview of the ECL options

sage.libs.ecl.test_sigint_before_ecl_sig_on()
2.1 Sage interface to Cremona's eclib library (also known as mwrank)

This is the Sage interface to John Cremona's eclib C++ library for arithmetic on elliptic curves. The classes defined in this module give Sage interpreter-level access to some of the functionality of eclib. For most purposes, it is not necessary to directly use these classes. Instead, one can create an EllipticCurve and call methods that are implemented using this module.

**Note:** This interface is a direct library-level interface to eclib, including the 2-descent program mwrank.

```python
class sage.libs.eclib.interface.mwrank_EllipticCurve(ainvs, verbose=False):
    Bases: sage.structure.sage_object.SageObject

    The mwrank_EllipticCurve class represents an elliptic curve using the Curvedata class from eclib, called here an 'mwrank elliptic curve'.

    Create the mwrank elliptic curve with invariants ainvs, which is a list of 5 or less integers \(a_1, a_2, a_3, a_4, a_5\).

    If strictly less than 5 invariants are given, then the first ones are set to 0, so, e.g., \([3, 4]\) means \(a_1 = a_2 = a_3 = 0\) and \(a_4 = 3, a_5 = 4\).

    **INPUT:**

    • `ainvs` (list or tuple) – a list of 5 or less integers, the coefficients of a nonsingular Weierstrass equation.
    • `verbose` (bool, default False) – verbosity flag. If True, then all Selmer group computations will be verbose.

    **EXAMPLES:**

    We create the elliptic curve \(y^2 + y = x^3 + x^2 - 2x\):

    ```python
    sage: e = mwrank_EllipticCurve([0, 1, 1, -2, 0])
    sage: e.ainvs()
    [0, 1, 1, -2, 0]
    ```

    This example illustrates that omitted \(a\)-invariants default to 0:

    ```python
    sage: e = mwrank_EllipticCurve([3, -4])
    sage: e
    y^2 = x^3 + 3*x - 4
    ```
```
sage: e.ainvs()
[0, 0, 0, 3, -4]

The entries of the input list are coerced to int. If this is impossible, then an error is raised:

sage: e = mwrank_EllipticCurve([3, -4.8]); e
Traceback (most recent call last):
...
TypeError: ainvs must be a list or tuple of integers.

When you enter a singular model you get an exception:

sage: e = mwrank_EllipticCurve([0, 0])
Traceback (most recent call last):
...
ArithmeticError: Invariants (= 0,0,0,0,0) do not describe an elliptic curve.

**CPS_height_bound()**

Return the Cremona-Prickett-Siksek height bound. This is a floating point number \( B \) such that if \( P \) is a point on the curve, then the naive logarithmic height \( h(P) \) is less than \( B + \hat{h}(P) \), where \( \hat{h}(P) \) is the canonical height of \( P \).

**Warning:** We assume the model is minimal!

**EXAMPLES:**

sage: E = mwrank_EllipticCurve([0, 0, 0, -1002231243161, 0])
sage: E.CPS_height_bound()
14.163198527061496
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: E.CPS_height_bound()
0.0

**ainvs()**

Returns the \( a \)-invariants of this mwrank elliptic curve.

**EXAMPLES:**

sage: E = mwrank_EllipticCurve([0,0,1,-1,0])
sage: E.ainvs()
[0, 0, 1, -1, 0]

certain()**

Returns True if the last \texttt{two_descent()} call provably correctly computed the rank. If \texttt{two_descent()} hasn’t been called, then it is first called by \texttt{certain()} using the default parameters.

The result is True if and only if the results of the methods \texttt{rank()} and \texttt{rank_bound()} are equal.

**EXAMPLES:**

A 2-descent does not determine \( E(Q) \) with certainty for the curve \( y^2 + y = x^3 - x^2 - 120x - 2183 \):

sage: E = mwrank_EllipticCurve([0, -1, 1, -120, -2183])
sage: E.two_descent(False)
The previous value is only a lower bound; the upper bound is greater:

```
sage: E.rank_bound()
sage: 2
```

In fact the rank of $E$ is actually 0 (as one could see by computing the $L$-function), but Sha has order 4 and the 2-torsion is trivial, so mwrank cannot conclusively determine the rank in this case.

**conductor()**

Return the conductor of this curve, computed using Cremona’s implementation of Tate’s algorithm.

**Note:** This is independent of PARI’s.

**EXAMPLES:**

```
sage: E = mwrank_EllipticCurve([1, 1, 0, -6958, -224588])
sage: E.conductor()
sage: 2310
```

**gens()**

Return a list of the generators for the Mordell-Weil group.

**EXAMPLES:**

```
sage: E = mwrank_EllipticCurve([0, 0, 1, -1, 0])
sage: E.gens()
[[0, -1, 1]]
```

**isogeny_class(verbos=False)**

Returns the isogeny class of this mwrank elliptic curve.

**EXAMPLES:**

```
sage: E = mwrank_EllipticCurve([0,-1,1,0,0])
sage: E.isogeny_class()
sage: (([0, -1, 1, 0, 0], [0, -1, 1, -10, -20], [0, -1, 1, -7820, -263580]),
    [[0, 5, → 0], [5, 0, 5], [0, 5, 0]])
```

**rank()**

Returns the rank of this curve, computed using `two_descent()`.

In general this may only be a lower bound for the rank; an upper bound may be obtained using the function `rank_bound()`. To test whether the value has been proved to be correct, use the method `certain()`.

**EXAMPLES:**

```
sage: E = mwrank_EllipticCurve([0, -1, 0, -900, -10098])
sage: E.rank()
sage: 0
sage: E.certain()
sage: True
```
rank_bound()  
Returns an upper bound for the rank of this curve, computed using two_descent().

If the curve has no 2-torsion, this is equal to the 2-Selmer rank. If the curve has 2-torsion, the upper bound may be smaller than the bound obtained from the 2-Selmer rank minus the 2-rank of the torsion, since more information is gained from the 2-isogenous curve or curves.

EXAMPLES:

The following is the curve 960D1, which has rank 0, but Sha of order 4:

```
sage: E = mwrank_EllipticCurve([0, -1, 0, -900, -10098])
sage: E.rank_bound()
0
sage: E.rank()
0
```

In this case the rank was computed using a second descent, which is able to determine (by considering a 2-isogenous curve) that Sha is nontrivial. If we deliberately stop the second descent, the rank bound is larger:

```
sage: E = mwrank_EllipticCurve([0, -1, 0, -900, -10098])
sage: E.two_descent(second_descent=False, verbose=False)
sage: E.rank_bound()
2
```

In contrast, for the curve 571A, also with rank 0 and Sha of order 4, we only obtain an upper bound of 2:

```
sage: E = mwrank_EllipticCurve([0, -1, 1, -929, -10595])
sage: E.rank_bound()
2
```

In this case the value returned by rank() is only a lower bound in general (though this is correct):

```
sage: E.rank()
0
sage: E.certain()
False
```

regulator()

Return the regulator of the saturated Mordell-Weil group.

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0, 0, 1, -1, 0])
sage: E.regulator()
0.05111140823996884
```

saturate(bound=-1)

Compute the saturation of the Mordell-Weil group at all primes up to bound.

INPUT:
• bound (int, default -1) – Use -1 (the default) to saturate at all primes, 0 for no saturation, or n (a positive integer) to saturate at all primes up to n.

EXAMPLES:
Since the 2-descent automatically saturates at primes up to 20, it is not easy to come up with an example where saturation has any effect:

```python
sage: E = mwrank_EllipticCurve([0, 0, 0, -1002231243161, 0])
sage: E.gens()
[[-1001107, -4004428, 1]]
sage: E.saturate()
sage: E.gens()
[[-1001107, -4004428, 1]]
```

Check that trac ticket #18031 is fixed:

```python
sage: E = mwrank_EllipticCurve([0, -1, 1, -266, 968])
sage: Q1 = E([-1995,3674,125])
sage: Q2 = E([157,1950,1])
sage: E.saturation([Q1,Q2])
([1 : -27 : 1], (157 : 1950 : 1)), 3, 0.801588644684981
```

### selmer_rank()
Returns the rank of the 2-Selmer group of the curve.

EXAMPLES:
The following is the curve 960D1, which has rank 0, but Sha of order 4. The 2-torsion has rank 2, and the Selmer rank is 3:

```python
sage: E = mwrank_EllipticCurve([0, -1, 0, -900, -10098])
sage: E.selmer_rank()
3
```

Nevertheless, we can obtain a tight upper bound on the rank since a second descent is performed which establishes the 2-rank of Sha:

```python
sage: E.rank_bound()
0
```

To show that this was resolved using a second descent, we do the computation again but turn off second_descent:

```python
sage: E = mwrank_EllipticCurve([0, -1, 0, -900, -10098])
sage: E.two_descent(second_descent = False, verbose=False)
sage: E.rank_bound()
2
```

For the curve 571A, also with rank 0 and Sha of order 4, but with no 2-torsion, the Selmer rank is strictly greater than the rank:

```python
sage: E = mwrank_EllipticCurve([0, -1, 1, -929, -10595])
sage: E.selmer_rank()
2
sage: E.rank_bound()
2
```

In cases like this with no 2-torsion, the rank upper bound is always equal to the 2-Selmer rank. If we ask for the rank, all we get is a lower bound:
```python
sage: E.rank()
0
sage: E.certain()
False
```

**set_verbose** *(verbose)*

Set the verbosity of printing of output by the `two_descent()` and other functions.

**INPUT:**

- `verbose` *(int)* – if positive, print lots of output when doing 2-descent.

**EXAMPLES:**

```python
sage: E = mwrank_EllipticCurve([0, 0, 1, -1, 0])
sage: E.saturate()  # no output
sage: E.gens()
[[0, -1, 1]]

sage: E = mwrank_EllipticCurve([0, 0, 1, -1, 0])
sage: E.set_verbose(1)
sage: E.saturate()  # tol 1e-10
```

Basic pair: I=48, J=-432
disc=255744
2-adic index bound = 2
By Lemma 5.1(a), 2-adic index = 1
2-adic index = 1
One (I,J) pair
Looking for quartics with I = 48, J = -432
Looking for Type 2 quartics:
Trying positive a from 1 up to 1 (square a first...)
(1,0,-6,4,1) --trivial
Trying positive a from 1 up to 1 (...then non-square a)
Finished looking for Type 2 quartics.
Looking for Type 1 quartics:
Trying positive a from 1 up to 2 (square a first...)
(1,0,0,4,4) --nontrivial...(x:y:z) = (1 : 1 : 0)
Point = [0:0:1]
height = 0.0511114082399688402358
Rank of B=im(eps) increases to 1 (The previous point is on the egg)
Exiting search for Type 1 quartics after finding one which is globally soluble.
Mordell rank contribution from B=im(eps) = 1
Seimler rank contribution from B=im(eps) = 1
Sha rank contribution from B=im(eps) = 0
Mordell rank contribution from A=ker(eps) = 0
Seimler rank contribution from A=ker(eps) = 0
Sha rank contribution from A=ker(eps) = 0
Searching for points (bound = 8)...done:
found points which generate a subgroup of rank 1
and regulator 0.0511114082399688402358
Processing points found during 2-descent...done:
now regulator = 0.0511114082399688402358
Saturating (with bound = -1)...done:
points were already saturated.
```

**silverman_bound()**

Return the Silverman height bound. This is a floating point number $B$ such that if $P$ is a point on the curve, then the naive logarithmic height $h(P)$ is less than $B + \hat{h}(P)$, where $\hat{h}(P)$ is the canonical height.
of $P$.

**Warning:** We assume the model is minimal!

**EXAMPLES:**

```python
sage: E = mwrank_EllipticCurve([0, 0, 0, -1002231243161, 0])
sage: E.silverman_bound()
18.29545210468247
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: E.silverman_bound()
6.284833369972403
```

**two_descent** *(verbose=True, selmer_only=False, first_limit=20, second_limit=8, n_aux=-1, second_descent=True)*

Compute 2-descent data for this curve.

**INPUT:**

- `verbose` (bool, default True) – print what mwrank is doing.
- `selmer_only` (bool, default False) – selmer_only switch.
- `first_limit` (int, default 20) – bound on $|x| + |z|$ in quartic point search.
- `second_limit` (int, default 8) – bound on $\log \max(|x|, |z|)$, i.e. logarithmic.
- `n_aux` (int, default -1) – (only relevant for general 2-descent when 2-torsion trivial) number of primes used for quartic search. $n_{aux}=-1$ causes default (8) to be used. Increase for curves of higher rank.
- `second_descent` (bool, default True) – (only relevant for curves with 2-torsion, where mwrank uses descent via 2-isogeny) flag determining whether or not to do second descent. Default strongly recommended.

**OUTPUT:**

Nothing – nothing is returned.

**class** `sage.libs.eclib.interface.mwrank_MordellWeil` *(curve, verbose=True, pp=1, maxr=999)*

**Bases:** `sage.structure.sage_object.SageObject`

The `mwrank_MordellWeil` class represents a subgroup of a Mordell-Weil group. Use this class to saturate a specified list of points on an `mwrank_EllipticCurve`, or to search for points up to some bound.

**INPUT:**

- `curve` (`mwrank_EllipticCurve`) – the underlying elliptic curve.
- `verbose` (bool, default False) – verbosity flag (controls amount of output produced in point searches).
- `pp` (int, default 1) – process points flag (if nonzero, the points found are processed, so that at all times only a $\mathbb{Z}$-basis for the subgroup generated by the points found so far is stored; if zero, no processing is done and all points found are stored).
- `maxr` (int, default 999) – maximum rank (quit point searching once the points found generate a subgroup of this rank; useful if an upper bound for the rank is already known).

**EXAMPLES:**
```python
sage: E = mwrank_EllipticCurve([1,0,1,4,-6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ
Subgroup of Mordell-Weil group: []
sage: EQ.search(2)
P1 = [0:1:0]  is torsion point, order 1
P1 = [1:-1:1] is torsion point, order 2
P1 = [2:2:1]  is torsion point, order 3
P1 = [9:23:1] is torsion point, order 6

sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.search(2)
P1 = [0:1:0]  is torsion point, order 1
P1 = [-3:0:1] is generator number 1
...
P4 = [-91:804:343] = -2*P1 + 2*P2 + 1*P3 (mod torsion)
sage: EQ
Subgroup of Mordell-Weil group: [[1:-1:1], [-2:3:1], [-14:25:8]]
```

Example to illustrate the verbose parameter:

```python
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E, verbose=False)
sage: EQ
Subgroup of Mordell-Weil group: [[1:-1:1], [-2:3:1], [-14:25:8]]

sage: EQ = mwrank_MordellWeil(E, verbose=True)
sage: EQ.search(1)
P1 = [0:1:0]  is torsion point, order 1
P1 = [-3:0:1] is generator number 1
saturating up to 20...Checking 2-saturation
Points have successfully been 2-saturated (max q used = 7)
Checking 3-saturation
Points have successfully been 3-saturated (max q used = 7)
Checking 5-saturation
Points have successfully been 5-saturated (max q used = 23)
Checking 7-saturation
Points have successfully been 7-saturated (max q used = 41)
Checking 11-saturation
Points have successfully been 11-saturated (max q used = 17)
Checking 13-saturation
Points have successfully been 13-saturated (max q used = 43)
Checking 17-saturation
Points have successfully been 17-saturated (max q used = 31)
Checking 19-saturation
Points have successfully been 19-saturated (max q used = 37)
done
P2 = [-2:3:1]  is generator number 2
saturating up to 20...Checking 2-saturation
possible kernel vector = [1,1]
This point may be in 2E(Q): [14:-52:1]
...and it is!
Replacing old generator #1 with new generator [1:-1:1]
Points have successfully been 2-saturated (max q used = 7)
Index gain = 2^1
```

(continues on next page)
Checking 3-saturation
Points have successfully been 3-saturated (max q used = 13)
Checking 5-saturation
Points have successfully been 5-saturated (max q used = 67)
Checking 7-saturation
Points have successfully been 7-saturated (max q used = 53)
Checking 11-saturation
Points have successfully been 11-saturated (max q used = 73)
Checking 13-saturation
Points have successfully been 13-saturated (max q used = 103)
Checking 17-saturation
Points have successfully been 17-saturated (max q used = 113)
Checking 19-saturation
Points have successfully been 19-saturated (max q used = 47)
   done (index = 2).
   Gained index 2, new generators = [ [1:-1:1] [-2:3:1] ]
P3 = [-14:25:8]  is generator number 3
   saturating up to 20...Checking 2-saturation
Points have successfully been 2-saturated (max q used = 11)
Checking 3-saturation
Points have successfully been 3-saturated (max q used = 13)
Checking 5-saturation
Points have successfully been 5-saturated (max q used = 71)
Checking 7-saturation
Points have successfully been 7-saturated (max q used = 101)
Checking 11-saturation
Points have successfully been 11-saturated (max q used = 127)
Checking 13-saturation
Points have successfully been 13-saturated (max q used = 151)
Checking 17-saturation
Points have successfully been 17-saturated (max q used = 139)
Checking 19-saturation
Points have successfully been 19-saturated (max q used = 179)
   done (index = 1).
P4 = [-1:3:1] = -1*P1 + -1*P2 + -1*P3 (mod torsion)
P4 = [0:2:1] = 2*P1 + 0*P2 + 1*P3 (mod torsion)
P4 = [2:13:8] = -3*P1 + 1*P2 + -1*P3 (mod torsion)
P4 = [1:0:1] = -1*P1 + 0*P2 + 0*P3 (mod torsion)
P4 = [2:0:1] = -1*P1 + 1*P2 + 0*P3 (mod torsion)
P4 = [18:7:8] = -2*P1 + -1*P2 + -1*P3 (mod torsion)
P4 = [3:3:1] = 1*P1 + 0*P2 + 1*P3 (mod torsion)
P4 = [4:6:1] = 0*P1 + -1*P2 + -1*P3 (mod torsion)
P4 = [36:69:64] = 1*P1 + -2*P2 + 0*P3 (mod torsion)
P4 = [68:-25:64] = -2*P1 + -1*P2 + -2*P3 (mod torsion)
P4 = [12:35:27] = 1*P1 + -1*P2 + -1*P3 (mod torsion)
\[\text{sage: } \text{EQ}\]
Subgroup of Mordell-Weil group: \([\text{[1:-1:1], [-2:3:1], [-14:25:8]}]\)

Example to illustrate the process points (\(pp\)) parameter:

\[\text{sage: } \text{E = mwrk\_EllipticCurve([0,0,1,-7,6])}\]
\[\text{sage: } \text{EQ = mwrk\_MordellWeil(E, verbose=False, pp=1)}\]
\[\text{sage: } \text{EQ.search(1)}; \text{EQ} \# \text{generators only}\]
Subgroup of Mordell-Weil group: \([\text{[1:-1:1], [-2:3:1], [-14:25:8]}]\)
\[\text{sage: } \text{EQ = mwrk\_MordellWeil(E, verbose=False, pp=0)}\]
\[\text{sage: } \text{EQ.search(1)}; \text{EQ} \# \text{all points found}\]
Subgroup of Mordell-Weil group: \([\text{[1:-1:1], [-2:3:1], [-14:25:8], [-1:3:1], [0:1:1], [1:0:1], [2:0:1], [18:7:8], [3:3:1], [4:6:1]]}\)
\[\text{sage: } \text{EQ.search(1); EQ} \# \text{more points found}\]
Subgroup of Mordell-Weil group: \([\text{[-3:0:1], [-2:3:1], [-14:25:8], [-1:3:1], [0:1:1], [2:0:1], [18:7:8], [3:3:1], [4:6:1]}\)
\[\text{sage: } \text{EQ.search(1); EQ} \# \text{all points found}\]
Subgroup of Mordell-Weil group: \([\text{[-3:0:1], [-2:3:1], [-14:25:8], [-1:3:1], [0:1:1], [2:0:1], [18:7:8], [3:3:1], [4:6:1]}\)

2.1. Sage interface to Cremona’s eclib library (also known as mwrk)
points()

Return a list of the generating points in this Mordell-Weil group.

OUTPUT:

(list) A list of lists of length 3, each holding the primitive integer coordinates \([x, y, z]\) of a generating point.

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.search(1)
P1 = [0:1:0] is torsion point, order 1
P1 = [-3:0:1] is generator number 1
...
P4 = [12:35:27] = 1*P1 + -1*P2 + -1*P3 (mod torsion)
sage: EQ.points()
[[1, -1, 1], [-2, 3, 1], [-14, 25, 8]]
```

process(v, sat=0)

This function allows one to add points to a \texttt{mwrank\textunderscore MordellWeil} object.

Process points in the list \(v\), with saturation at primes up to \(sat\). If \(sat\) is zero (the default), do no saturation.

INPUT:

- \(v\) (list of 3-tuples or lists of ints or Integers) – a list of triples of integers, which define points on the curve.
- \(sat\) (int, default 0) – saturate at primes up to \(sat\), or at all primes if \(sat\) is zero.

OUTPUT:

None. But note that if the \texttt{verbose} flag is set, then there will be some output as a side-effect.

EXAMPLES:

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: E.gens()
[[1, -1, 1], [-2, 3, 1], [-14, 25, 8]]
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.process([[1, -1, 1], [-2, 3, 1], [-14, 25, 8]], sat=20)
P1 = [1:-1:1] is generator number 1
...
```

Example to illustrate the saturation parameter \(sat\):

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.process([[1547,-2967,343], [2707496766203306, 864581029138191, ˓→296971514023272], [-13422227300, -49322830557, 12167000000]], sat=20)
P1 = [1547:-2967:343] is generator number 1
...
```
Here the processing was followed by saturation at primes up to 20. Now we prevent this initial saturation:

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.process([(1547,-2967,343), (2707496766203306, 864581029138191, 2969715140223272), [-13422227300, -49322830557, 12167000000]], sat=0)
P1 = [1547;-2967;343] is generator number 1
P2 = [2707496766203306;864581029138191;2969715140223272] is generator number 2
P3 = [-13422227300;-49322830557;12167000000] is generator number 3
sage: EQ.points()
[[1547, -2967, 343], [2707496766203306, 864581029138191, 2969715140223272], [-13422227300, -49322830557, 12167000000]]
sage: EQ.regulator()
375.4292028825455
sage: EQ.saturate(2) # points were not 2-saturated
saturating basis...Saturation index bound = 93
WARNING: saturation at primes p > 2 will not be done;
...
Gained index 2
New regulator = 93.857...
(False, 2, '[]')
sage: EQ.points()
[[1547, -2967, 343], [2707496766203306, 864581029138191, 2969715140223272], [-13422227300, -49322830557, 12167000000]]
sage: EQ.regulator()
93.85730072063639
sage: EQ.saturate(3) # points were not 3-saturated
saturating basis...Saturation index bound = 46
WARNING: saturation at primes p > 3 will not be done;
...
Gained index 3
New regulator = 10.428...
(False, 3, '[]')
sage: EQ.points()
[[1547, -2967, 343], [2707496766203306, 864581029138191, 2969715140223272], [-13422227300, -49322830557, 12167000000]]
sage: EQ.regulator()
10.4285889689596
sage: EQ.saturate(5) # points were not 5-saturated
saturating basis...Saturation index bound = 15
WARNING: saturation at primes p > 5 will not be done;
...
Gained index 5
New regulator = 0.417...
(False, 5, '[]')
sage: EQ.points()
[[1547, -2967, 343], [2707496766203306, 864581029138191, 2969715140223272], [-13422227300, -49322830557, 12167000000]]
sage: EQ.regulator()
0.417143558758384
sage: EQ.saturate() # points are now saturated
saturating basis...Saturation index bound = 3
Checking saturation at [2 3]
```

Gained index 5, new generators = [ [-2:3:1] [-14:25:8] [1:-1:1] ]
Checking 2-saturation
Points were proved 2-saturated (max q used = 11)
Checking 3-saturation
Points were proved 3-saturated (max q used = 13)
done
(True, 1, '[]')

\begin{verbatim}
rank()
Return the rank of this subgroup of the Mordell-Weil group.

OUTPUT:
(int) The rank of this subgroup of the Mordell-Weil group.

EXAMPLES:

\begin{verbatim}
sage: E = mwrank_EllipticCurve([0,-1,1,0,0])
sage: E.rank()
0
\end{verbatim}

A rank 3 example:

\begin{verbatim}
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.rank()
0
sage: EQ.regulator()
1.0
\end{verbatim}

The preceding output is correct, since we have not yet tried to find any points on the curve either by searching or 2-descent:

\begin{verbatim}
sage: EQ
Subgroup of Mordell-Weil group: []
\end{verbatim}

Now we do a very small search:

\begin{verbatim}
sage: EQ.search(1)
P1 = [0:1:0] is torsion point, order 1
P1 = [-3:0:1] is generator number 1
saturating up to 20...Checking 2-saturation
...P4 = [12:35:27] = 1*P1 + -1*P2 + -1*P3 (mod torsion)
sage: EQ
Subgroup of Mordell-Weil group: [[1:-1:1], [-2:3:1], [-14:25:8]]
sage: EQ.rank()
3
sage: EQ.regulator()
0.417143558758384
\end{verbatim}

We do in fact now have a full Mordell-Weil basis.

\begin{verbatim}
regulator()
Return the regulator of the points in this subgroup of the Mordell-Weil group.
\end{verbatim}

\textbf{Note:} \texttt{eclib} can compute the regulator to arbitrary precision, but the interface currently returns the output as a \texttt{float}.
OUTPUT:

(float) The regulator of the points in this subgroup.

EXAMPLES:

```python
sage: E = mwrank_EllipticCurve([0,-1,1,0,0])
sage: E.regulator()
1.0
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: E.regulator()
0.417143558758384
```

`saturate (max_prime=-1, odd_primes_only=False)`

Saturate this subgroup of the Mordell-Weil group.

INPUT:

- `max_prime` (int, default -1) – saturation is performed for all primes up to `max_prime`. If -1 (the default), an upper bound is computed for the primes at which the subgroup may not be saturated, and this is used; however, if the computed bound is greater than a value set by the `eclib` library (currently 97) then no saturation will be attempted at primes above this.
- `odd_primes_only` (bool, default False) – only do saturation at odd primes. (If the points have been found via `two_descent()` they should already be 2-saturated.)

OUTPUT:

(3-tuple) `(ok, index, unsatlist)` where:

- `ok` (bool) – True if and only if the saturation was provably successful at all primes attempted. If the default was used for `max_prime` and no warning was output about the computed saturation bound being too high, then True indicates that the subgroup is saturated at all primes.
- `index` (int) – the index of the group generated by the original points in their saturation.
- `unsatlist` (list of ints) – list of primes at which saturation could not be proved or achieved. Increasing the precision should correct this, since it happens when a linear combination of the points appears to be a multiple of \( p \) but cannot be divided by \( p \). (Note that `eclib` uses floating point methods based on elliptic logarithms to divide points.)

**Note:** We emphasize that if this function returns `True` as the first return argument (`ok`), and if the default was used for the parameter `max_prime`, then the points in the basis after calling this function are saturated at all primes, i.e., saturating at the primes up to `max_prime` are sufficient to saturate at all primes. Note that the function might not have needed to saturate at all primes up to `max_prime`. It has worked out what prime you need to saturate up to, and that prime might be smaller than `max_prime`.

**Note:** Currently (May 2010), this does not remember the result of calling `search()`. So calling `search()` up to height 20 then calling `saturate()` results in another search up to height 18.

EXAMPLES:

```python
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
```

We initialise with three points which happen to be 2, 3 and 5 times the generators of this rank 3 curve. To prevent automatic saturation at this stage we set the parameter `sat` to 0 (which is in fact the default):
Now we saturate at $p = 2$, and gain index 2:

```python
sage: EQ.saturate(2) # points were not 2-saturated
saturating basis...Saturation index bound = 93
WARNING: saturation at primes $p > 2$ will not be done;
...
Gained index 2
New regulator = 93.857...
(False, 2, '[ ]')
sage: EQ
Subgroup of Mordell-Weil group: [[-2:3:1], [-14:25:8], [-13422227300:-49322830557:12167000000]]
sage: EQ.regulator()
93.85730072063639
```

Now we saturate at $p = 3$, and gain index 3:

```python
sage: EQ.saturate(3) # points were not 3-saturated
saturating basis...Saturation index bound = 46
WARNING: saturation at primes $p > 3$ will not be done;
...
Gained index 3
New regulator = 10.428...
(False, 3, '[ ]')
sage: EQ
Subgroup of Mordell-Weil group: [[-2:3:1], [-14:25:8], [-13422227300:-49322830557:12167000000]]
sage: EQ.regulator()
10.4285889689596
```

Now we saturate at $p = 5$, and gain index 5:

```python
sage: EQ.saturate(5) # points were not 5-saturated
saturating basis...Saturation index bound = 15
WARNING: saturation at primes $p > 5$ will not be done;
...
Gained index 5
New regulator = 0.417...
(False, 5, '[ ]')
sage: EQ
Subgroup of Mordell-Weil group: [[-2:3:1], [-14:25:8], [1:-1:1]]
sage: EQ.regulator()
0.417143558758384
```
Finally we finish the saturation. The output here shows that the points are now provably saturated at all primes:

```python
sage: EQ.saturate()  # points are now saturated
saturating basis...Saturation index bound = 3
Checking saturation at [ 2 3 ]
Checking 2-saturation
Points were proved 2-saturated (max q used = 11)
Checking 3-saturation
Points were proved 3-saturated (max q used = 13)
done
(True, 1, '[]')
```

Of course, the `process()` function would have done all this automatically for us:

```python
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.process([[1547, -2967, 343], [2707496766203306, 864581029138191, ˓→2969715140223272], [-13422227300, -49322830557, 12167000000]], sat=5)
P1 = [1547:-2967:343] is generator number 1
...
Gained index 5, new generators = [ [-2:3:1] [-14:25:8] [1:-1:1] ]
sage: EQ
Subgroup of Mordell-Weil group: [[-2:3:1], [-14:25:8], [1:-1:1]]
sage: EQ.regulator()
0.417143558758384
```

But we would still need to use the `saturate()` function to verify that full saturation has been done:

```python
sage: EQ.saturate()  # points are now saturated
saturating basis...Saturation index bound = 3
Checking saturation at [ 2 3 ]
Checking 2-saturation
Points were proved 2-saturated (max q used = 11)
Checking 3-saturation
Points were proved 3-saturated (max q used = 13)
done
(True, 1, '[]')
```

Note the output of the preceding command: it proves that the index of the points in their saturation is at most 3, then proves saturation at 2 and at 3, by reducing the points modulo all primes of good reduction up to 11, respectively 13.

`search` *(height_limit=18, verbose=False)*

Search for new points, and add them to this subgroup of the Mordell-Weil group.

**INPUT:**

- `height_limit` *(float, default: 18) – search up to this logarithmic height.*

**Note:** On 32-bit machines, this *must* be < 21.48 else \(\exp(h_{\text{lim}}) > 2^{31}\) and overflows. On 64-bit machines, it must be at *most* 43.668. However, this bound is a logarithmic bound and increasing it by just 1 increases the running time by (roughly) \(\exp(1.5) = 4.5\), so searching up to even 20 takes a very long time.

**Note:** The search is carried out with a quadratic sieve, using code adapted from a version of Michael Stoll's `ratpoints` program. It would be preferable to use a newer version of `ratpoints`.

---

**2.1. Sage interface to Cremona’s eclib library (also known as mwrank)**

25
• `verbose (bool, default False)` – turn verbose operation on or off.

EXAMPLES:

A rank 3 example, where a very small search is sufficient to find a Mordell-Weil basis:

```python
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.search(1)
```

```
P1 = [0:1:0] is torsion point, order 1
P1 = [-3:0:1] is generator number 1
...
P4 = [12:35:27] = 1*P1 + -1*P2 + -1*P3 (mod torsion)
sage: EQ
Subgroup of Mordell-Weil group: [[1:-1:1], [-2:3:1], [-14:25:8]]
```

In the next example, a search bound of 12 is needed to find a non-torsion point:

```python
sage: E = mwrank_EllipticCurve([0, -1, 0, -18392, -1186248])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.search(11); EQ
```

```
P1 = [0:1:0] is torsion point, order 1
P1 = [161:0:1] is torsion point, order 2
Subgroup of Mordell-Weil group: []
sage: EQ.search(12); EQ
```

```
P1 = [0:1:0] is torsion point, order 1
P1 = [161:0:1] is torsion point, order 2
P1 = [4413270:10381877:27000] is generator number 1
...
Subgroup of Mordell-Weil group: [[4413270:10381877:27000]]
```

### 2.2 Cython interface to Cremona’s eclib library (also known as mwrank)

EXAMPLES:

```python
sage: from sage.libs.eclib.mwrank import _Curvedata, _mw
sage: c = _Curvedata(1,2,3,4,5)
```

```
sage: print(c)
[1,2,3,4,5]
b2 = 9   b4 = 11   b6 = 29   b8 = 35
c4 = -183  c6 = -3429
disc = -10351  (# real components = 1)
#torsion not yet computed
```

```python
sage: t = _mw(c)
sage: t.search(10)
sage: t
```

```
[[1:2:1]]
sage.libs.eclib.mwrank.get_precision()
Returns the working floating point bit precision of mwrank, which is equal to the global NTL real number precision.
```

OUTPUT:
(int) The current precision in bits.

See also `set_precision()`.

EXAMPLES:

```python
sage: mwrank_get_precision()
150
```

`sage.libs.eclib.mwrank.initprimes (filename, verb=False)`
Initialises mwrank/eclib’s internal prime list.

INPUT:

- `filename` (string) – the name of a file of primes.
- `verb` (bool: default False) – verbose or not?

EXAMPLES:

```python
sage: file = os.path.join(SAGE_TMP, 'PRIMES')
sage: with open(file, 'w') as fobj:
....:     _ = fobj.write(' '.join([str(p) for p in prime_range(10^7,10^7+20)]))
sage: mwrank_initprimes(file, verb=True)
Computed 78519 primes, largest is 1000253
reading primes from file ... 
read extra prime 10000019
finished reading primes from file ... 
Extra primes in list: 10000019

sage: mwrank_initprimes("x" + file, True)
Traceback (most recent call last):
... 
IOError: No such file or directory: ...
```

`sage.libs.eclib.mwrank.set_precision (n)`
Sets the working floating point bit precision of mwrank, which is equal to the global NTL real number precision.

NTL real number bit precision. This has a massive effect on the speed of mwrank calculations. The default (used if this function is not called) is n=150, but it might have to be increased if a computation fails.

INPUT:

- `n` – a positive integer: the number of bits of precision.

**Warning:** This change is global and affects all future calls of eclib functions by Sage.

**Note:** The minimal value to which the precision may be set is 53. Lower values will be increased to 53.

See also `get_precision()`.

EXAMPLES:

```python
sage: from sage.libs.eclib.mwrank import set_precision, get_precision
sage: old_prec = get_precision(); old_prec
150
sage: set_precision(50)
sage: get_precision()
50
```

(continues on next page)
2.3 Cremona matrices

class sage.libs.eclib.mat.Matrix
Bases: object

A Cremona Matrix.

EXAMPLES:

sage: M = CremonaModularSymbols(225)
sage: t = M.hecke_matrix(2)
sage: type(t)
<type 'sage.libs.eclib.mat.Matrix'>
sage: t
61 x 61 Cremona matrix over Rational Field

add_scalar(s)
Return new matrix obtained by adding s to each diagonal entry of self.

EXAMPLES:

sage: M = CremonaModularSymbols(23, cuspidal=True, sign=1)
sage: t = M.hecke_matrix(2); print(t.str())
[ 0 1]
[ 1 -1]
sage: w = t.add_scalar(3); print(w.str())
[3 1]
[1 2]

charpoly(var='x')
Return the characteristic polynomial of this matrix, viewed as as a matrix over the integers.

ALGORITHM:

Note that currently, this function converts this matrix into a dense matrix over the integers, then calls the charpoly algorithm on that, which I think is LinBox’s.

EXAMPLES:

sage: M = CremonaModularSymbols(33, cuspidal=True, sign=1)
sage: t = M.hecke_matrix(2)
sage: t.charpoly()
x^3 + 3*x^2 - 4
sage: t.charpoly().factor()
(x - 1) * (x + 2)^2

ncols()
Return the number of columns of this matrix.

EXAMPLES:
sage: M = CremonaModularSymbols(1234, sign=1)
sage: t = M.hecke_matrix(3); t.ncols()
156
sage: M.dimension()
156

\texttt{nrows()}

Return the number of rows of this matrix.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: M = CremonaModularSymbols(19, sign=1)
sage: t = M.hecke_matrix(13); t
2 x 2 Cremona matrix over Rational Field
sage: t.nrows()
2
\end{verbatim}

\texttt{sage_matrix_over_ZZ(sparse=True)}

Return corresponding Sage matrix over the integers.

\textbf{INPUT:}

- \texttt{sparse} – (default: \texttt{True}) whether the return matrix has a sparse representation

\textbf{EXAMPLES:}

\begin{verbatim}
sage: M = CremonaModularSymbols(23, cuspidal=True, sign=1)
sage: t = M.hecke_matrix(2)
sage: s = t.sage_matrix_over_ZZ(); s
[ 0 1]
[ 1 -1]
sage: type(s)
<type 'sage.matrix.matrix_integer_sparse.Matrix_integer_sparse'>
sage: s = t.sage_matrix_over_ZZ(sparse=False); s
[ 0 1]
[ 1 -1]
sage: type(s)
<type 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
\end{verbatim}

\texttt{str()}

Return full string representation of this matrix, never in compact form.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: M = CremonaModularSymbols(22, sign=1)
sage: t = M.hecke_matrix(13)
sage: t.str()
'\begin{bmatrix}
14 & 0 & 0 & 0 & 0
-4 & 12 & 0 & 8 & 4
0 & -6 & 4 & -6 & 0
4 & 2 & 0 & 6 & -4
0 & 0 & 0 & 14
\end{bmatrix}'
\end{verbatim}

\texttt{class sage.libs.eclib.mat.MatrixFactory}

\texttt{Bases: object}

\section*{2.4 Modular symbols using eclib newforms}

\texttt{class sage.libs.eclib.newforms.ECModularSymbol}

\texttt{Bases: object}
Modular symbol associated with an elliptic curve, using John Cremona's newforms class.

EXAMPLES:

```python
sage: from sage.libs.eclib.newforms import ECModularSymbol
sage: E = EllipticCurve('11a')
sage: M = ECModularSymbol(E,1); M
Modular symbol with sign 1 over Rational Field attached to Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
```

By default, symbols are based at the cusp $\infty$, i.e. we evaluate $\{\infty, r\}$:

```python
sage: [M(1/i) for i in range(1,11)]
[2/5, -8/5, -3/5, 7/5, 12/5, 12/5, 7/5, -3/5, -8/5, 2/5]
```

We can also switch the base point to the cusp 0:

```python
sage: [M(1/i, base_at_infinity=False) for i in range(1,11)]
[0, -2, -1, 1, 2, 2, 1, -1, -2, 0]
```

For the minus symbols this makes no difference since $\{0, \infty\}$ is in the plus space. Note that to evaluate minus symbols the space must be defined with sign 0, which makes both signs available:

```python
sage: M = ECModularSymbol(E,0); M
Modular symbol with sign 0 over Rational Field attached to Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: [M(1/i, -1) for i in range(1,11)]
[0, 0, 1, 1, 0, 0, -1, -1, 0, 0]
```

If the ECModularSymbol is created with sign 0 then as well as asking for both + and - symbols, we can also obtain both (as a tuple). However it is more work to create the full modular symbol space:

```python
sage: E = EllipticCurve('11a1')
sage: M = ECModularSymbol(E,0); M
Modular symbol with sign 0 over Rational Field attached to Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: [M(1/i) for i in range(2,11)]
[[-8/5, 0], [-3/5, 1], [7/5, 1], [12/5, 0], [12/5, 0], [7/5, -1], [-3/5, -1], [-8/5, 0], [2/5, 0]]
```

The curve is automatically converted to its minimal model:

```python
sage: E = EllipticCurve([0,0,0,0,1/4])
sage: ECModularSymbol(E)
Modular symbol with sign 1 over Rational Field attached to Elliptic Curve defined by y^2 + y = x^3 over Rational Field
```

Non-optimal curves are handled correctly in eclib, by comparing the ratios of real and/or imaginary periods:
```python
sage: from sage.libs.eclib.newforms import ECModularSymbol
sage: E1 = EllipticCurve('11a1')  # optimal
sage: E1.period_lattice().basis()
(1.26920930427955, 0.634604652139777 + 1.45881661693850*I)
sage: M1 = ECModularSymbol(E1,0)
sage: M1(0)
[2/5, 0]
sage: M1(1/3)
[-3/5, 1]

One non-optimal curve has real period 1/5 that of the optimal one, so plus symbols scale up by a factor of 5
while minus symbols are unchanged:

```python
sage: E2 = EllipticCurve('11a2')  # not optimal
sage: E2.period_lattice().basis()
(0.253841860855911, 0.126920930427955 + 1.45881661693850*I)
sage: M2 = ECModularSymbol(E2,0)
sage: M2(0)
[2, 0]
sage: M2(1/3)
[-3, 1]
sage: all((M2(r,1)==5*M1(r,1)) for r in QQ.range_by_height(10))
True
sage: all((M2(r,-1)==M1(r,-1)) for r in QQ.range_by_height(10))
True

The other non-optimal curve has real period 5 times that of the optimal one, so plus symbols scale down by a
factor of 5; again, minus symbols are unchanged:

```python
sage: E3 = EllipticCurve('11a3')  # not optimal
sage: E3.period_lattice().basis()
(6.34604652139777, 3.17302326069888 + 1.45881661693850*I)
sage: M3 = ECModularSymbol(E3,0)
sage: M3(0)
[2/25, 0]
sage: M3(1/3)
[-3/25, 1]
sage: all((5*M3(r,1)==M1(r,1)) for r in QQ.range_by_height(10))
True
sage: all((M3(r,-1)==M1(r,-1)) for r in QQ.range_by_height(10))
True
```

2.5 Cremona modular symbols

```python
class sage.libs.eclib.homspace.ModularSymbols
    Bases: object
    
    Class of Cremona Modular Symbols of given level and sign (and weight 2).

    EXAMPLES:

```python
sage: M = CremonaModularSymbols(225)
sage: type(M)
<type 'sage.libs.eclib.homspace.ModularSymbols'>
```
dimension()
    Return the dimension of this modular symbols space.

EXAMPLES:

    sage: M = CremonaModularSymbols(1234, sign=1)
    sage: M.dimension()
    156

hecke_matrix(p, dual=False, verbose=False)
    Return the matrix of the p-th Hecke operator acting on this space of modular symbols.

    The result of this command is not cached.

    INPUT:

        • p – a prime number
        • dual – (default: False) whether to compute the Hecke operator acting on the dual space, i.e., the transpose of the Hecke operator
        • verbose – (default: False) print verbose output

    OUTPUT:

        (matrix) If p divides the level, the matrix of the Atkin-Lehner involution \( W_p \) at p; otherwise the matrix of the Hecke operator \( T_p \).

EXAMPLES:

    sage: M = CremonaModularSymbols(37)
    sage: t = M.hecke_matrix(2); t
    5 x 5 Cremona matrix over Rational Field
    sage: print(t.str())
    [ 3 0 0 0 0]
    [-1 -1 1 1 0]
    [ 0 0 -1 0 1]
    [-1 1 0 -1 -1]
    [ 0 0 1 0 -1]
    sage: t.charpoly().factor()
    (x - 3) * x^2 * (x + 2)^2
    sage: print(M.hecke_matrix(2, dual=True).str())
    [ 3 -1 0 -1 0]
    [ 0 -1 0 1 0]
    [ 0 1 -1 0 1]
    [ 0 1 0 -1 0]
    [ 0 0 1 -1 -1]
    sage: w = M.hecke_matrix(37); w
    5 x 5 Cremona matrix over Rational Field
    sage: w.charpoly().factor()
    (x - 1)^2 * (x + 1)^3
    sage: sw = w.sage_matrix_over_ZZ()
    sage: st = t.sage_matrix_over_ZZ()
    sage: sw^2 == sw.parent()(1)
    True
    sage: st*sw == sw*st
    True

is_cuspidal()
    Return whether or not this space is cuspidal.

EXAMPLES:
sage: M = CremonaModularSymbols(1122); M.is_cuspidal()
0
sage: M = CremonaModularSymbols(1122, cuspidal=True); M.is_cuspidal()
1

**level()**

Return the level of this modular symbols space.

EXAMPLES:

```python
sage: M = CremonaModularSymbols(1234, sign=1)
sage: M.level()
1234
```

**number_of_cusps()**

Return the number of cusps for $\Gamma_0(N)$, where $N$ is the level.

EXAMPLES:

```python
sage: M = CremonaModularSymbols(225)
sage: M.number_of_cusps()
24
```

**sign()**

Return the sign of this Cremona modular symbols space. The sign is either 0, +1 or -1.

EXAMPLES:

```python
sage: M = CremonaModularSymbols(1122, sign=1); M
Cremona Modular Symbols space of dimension 224 for Gamma_0(1122) of weight 2
-> with sign 1
sage: M.sign()
1
sage: M = CremonaModularSymbols(1122); M
Cremona Modular Symbols space of dimension 433 for Gamma_0(1122) of weight 2
-> with sign 0
sage: M.sign()
0
sage: M = CremonaModularSymbols(1122, sign=-1); M
Cremona Modular Symbols space of dimension 209 for Gamma_0(1122) of weight 2
-> with sign -1
sage: M.sign()
-1
```

**sparse_hecke_matrix** ($p$, dual=False, verbose=False, base_ring='ZZ')

Return the matrix of the $p$-th Hecke operator acting on this space of modular symbols as a sparse Sage matrix over base_ring. This is more memory-efficient than creating a Cremona matrix and then applying sage_matrix_over_ZZ with sparse=True.

The result of this command is not cached.

INPUT:

- $p$ – a prime number
- **dual** – (default: False) whether to compute the Hecke operator acting on the dual space, i.e., the transpose of the Hecke operator
- **verbose** – (default: False) print verbose output
(matrix) If \( p \) divides the level, the matrix of the Atkin-Lehner involution \( W_p \) at \( p \); otherwise the matrix of the Hecke operator \( T_p \).

EXAMPLES:

```python
sage: M = CremonaModularSymbols(37)
sage: t = M.sparse_hecke_matrix(2); type(t)
<type 'sage.matrix.matrix_integer_sparse.Matrix_integer_sparse'>
sage: print(t)
[ 3 0 0 0 0]
[-1 -1 1 1 0]
[ 0 0 -1 0 1]
[-1 1 0 -1 -1]
[ 0 0 1 0 -1]
sage: M = CremonaModularSymbols(5001)
sage: T = M.sparse_hecke_matrix(2)
sage: U = M.hecke_matrix(2).sage_matrix_over_ZZ(sparse=True)
sage: print(T == U)
True
sage: T = M.sparse_hecke_matrix(2, dual=True)
sage: print(T == U.transpose())
True
sage: T = M.sparse_hecke_matrix(2, base_ring=GF(7))
sage: print(T == U.change_ring(GF(7)))
True
```

This concerns an issue reported on trac ticket #21303:

```python
sage: C = CremonaModularSymbols(45, cuspidal=True, sign=-1)
sage: T2a = C.hecke_matrix(2).sage_matrix_over_ZZ()
sage: T2b = C.sparse_hecke_matrix(2)
sage: print(T2a == T2b)
True
```

## 2.6 Cremona modular symbols

`sage.libs.eclib.constructor.CremonaModularSymbols(level, sign=0, cuspidal=False, verbose=0)`

Return the space of Cremona modular symbols with given level, sign, etc.

**INPUT:**

- \( \text{level} \) – an integer >= 2 (at least 2, not just positive!)
- \( \text{sign} \) – an integer either 0 (the default) or 1 or -1.
- \( \text{cuspidal} \) – (default: False); if True, compute only the cuspidal subspace
- \( \text{verbose} \) – (default: False): if True, print verbose information while creating space

**EXAMPLES:**

```python
sage: M = CremonaModularSymbols(43); M
Cremona Modular Symbols space of dimension 7 for Gamma_0(43) of weight 2 with \( \rightarrow \) sign 0
sage: M = CremonaModularSymbols(43, sign=1); M
(continues on next page)
```
Cremona Modular Symbols space of dimension 4 for Gamma_0(43) of weight 2 with sign 1

sage: M = CremonaModularSymbols(43, cuspidal=True); M

Cremona Cuspidal Modular Symbols space of dimension 6 for Gamma_0(43) of weight 2 with sign 0

sage: M = CremonaModularSymbols(43, cuspidal=True, sign=1); M

Cremona Cuspidal Modular Symbols space of dimension 3 for Gamma_0(43) of weight 2 with sign 1

When run interactively, the following command will display verbose output:

sage: M = CremonaModularSymbols(43, verbose=1)

After 2-term relations, ngens = 22
ngens = 22
maxnumrel = 32
relation matrix has = 704 entries...
Finished 3-term relations: numrel = 16 ( maxnumrel = 32)
reimathas 42 nonzero entries (density = 0.0596591)
Computing kernel...
time to compute kernel = (... seconds)

rk = 7
Number of cusps is 2
ncusps = 2
About to compute matrix of delta
delta matrix done: size 2x7.
About to compute kernel of delta
done
Finished constructing homspace.

sage: M

Cremona Modular Symbols space of dimension 7 for Gamma_0(43) of weight 2 with sign 0

The input must be valid or a ValueError is raised:

sage: M = CremonaModularSymbols(-1)

Traceback (most recent call last):
... ...
ValueError: the level (= -1) must be at least 2

sage: M = CremonaModularSymbols(0)

Traceback (most recent call last):
... ...
ValueError: the level (= 0) must be at least 2

The sign can only be 0 or 1 or -1:

sage: M = CremonaModularSymbols(10, sign = -2)

Traceback (most recent call last):
... ...
ValueError: sign (= -2) is not supported; use 0, +1 or -1

We do allow -1 as a sign (see trac ticket #9476):

sage: CremonaModularSymbols(10, sign = -1)

Cremona Modular Symbols space of dimension 0 for Gamma_0(10) of weight 2 with sign -1

2.6. Cremona modular symbols
3.1 Flint imports

```python
sage.libs.flint.flint.free_flint_stack()
```

3.2 FLINT fmpz_poly class wrapper

AUTHORS:

- William Stein (2007-10-02) update for new flint; add arithmetic and creation of coefficients of arbitrary size.

```python
class sage.libs.flint.fmpz_poly.Fmpz_poly
    Bases: sage.structure.sage_object.SageObject

    Construct a new fmpz_poly from a sequence, constant coefficient, or string (in the same format as it prints).

    EXAMPLES:

    ```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: Fmpz_poly([1,2,3])
3 1 2 3
sage: Fmpz_poly(5)
1 5
sage: Fmpz_poly(str(Fmpz_poly([3,5,7])))
3 3 5 7
```
```

degree()

The degree of self.

EXAMPLES:

```python
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([1,2,3]); f
3 1 2 3
sage: f.degree()
2
sage: Fmpz_poly(range(1000)).degree()
999
sage: Fmpz_poly([2,0]).degree()
0
```
**derivative()**
Return the derivative of self.

**EXAMPLES:**
```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([1,2,6])
sage: f.derivative().list() == [2, 12]
True
```

**div_rem(other)**
Return self / other, self, % other.

**EXAMPLES:**
```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([1,3,4,5])
sage: g = f^23
sage: g.div_rem(f)[1]
0
sage: g.div_rem(f)[0] - f^22
0
sage: f = Fmpz_poly([1..10])
sage: g = Fmpz_poly([1,3,5])
sage: q, r = f.div_rem(g)
sage: q*f+r
17 1 2 3 4 4 4 10 11 17 18 22 26 30 23 26 18 20
sage: g
3 1 3 5
sage: q*g+r
10 1 2 3 4 5 6 7 8 9 10
```

**left_shift(n)**
Left shift self by n.

**EXAMPLES:**
```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([1,2])
sage: f.left_shift(1).list() == [0,1,2]
True
```

**list()**
Return self as a list of coefficients, lowest terms first.

**EXAMPLES:**
```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([2,1,0,-1])
sage: f.list()
[2, 1, 0, -1]
```

**pow_truncate(exp, n)**
Return self raised to the power of exp mod x^n.

**EXAMPLES:**
```
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([2,1,0,-1])
sage: f.list()
[2, 1, 0, -1]
```
pseudo_div(other)

pseudo_div_rem(other)

right_shift(n)

Right shift self by n.

EXAMPLES:

```python
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([1,2])
```

truncate(n)

Return the truncation of self at degree n.

EXAMPLES:

```python
sage: from sage.libs.flint.fmpz_poly import Fmpz_poly
sage: f = Fmpz_poly([1,1])
sage: g = f**10; g
11 1 10 45 120 210 252 210 120 45 10 1
sage: g.truncate(5)
5 1 10 45 120 210
```

### 3.3 FLINT Arithmetic Functions

sage.libs.flint.arith.bell_number(n)

Return the $n$-th Bell number.

See Wikipedia article Bell_number.

EXAMPLES:

```python
sage: from sage.libs.flint.arith import bell_number
sage: [bell_number(i) for i in range(10)]
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]
sage: bell_number(10)
115975
```

sage.libs.flint.arith.bernoulli_number(n)

Return the $n$-th Bernoulli number.

See Wikipedia article Bernoulli_number.

EXAMPLES:
Sage Reference Manual: C/C++ Library Interfaces, Release 8.7

```python
sage: from sage.libs.flint.arith import bernoulli_number
sage: [bernoulli_number(i) for i in range(10)]
[1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, -1/30, 0]
sage: bernoulli_number(10)
5/66
sage: bernoulli_number(40)
-261082718496449122051/13530
sage: bernoulli_number(100)
-94598037819122125295227433069493721872702841533066936133385696204311395415197247711/
33330
```

```python
sage.libs.flint.arith.dedekind_sum(p, q)
Return the Dedekind sum $s(p, q)$ where $p$ and $q$ are arbitrary integers.

See Wikipedia article Dedekind_sum.

EXAMPLES:

```python
sage: from sage.libs.flint.arith import dedekind_sum
sage: dedekind_sum(4, 5)
-1/5
```

```python
sage.libs.flint.arith.euler_number(n)
Return the Euler number of index $n$.

See Wikipedia article Euler_number.

EXAMPLES:

```python
sage: from sage.libs.flint.arith import euler_number
sage: [euler_number(i) for i in range(8)]
[1, 0, -1, 0, 5, 0, -61, 0]
```

```python
sage.libs.flint.arith.harmonic_number(n)
Return the harmonic number $H_n$.

See Wikipedia article Harmonic_number.

EXAMPLES:

```python
sage: from sage.libs.flint.arith import harmonic_number
sage: n = 500 + randint(0,500)
sage: bool( sum(1/k for k in range(1,n+1)) == harmonic_number(n) )
True
```

```python
sage.libs.flint.arith.number_of_partitions(n)
Return the number of partitions of the integer $n$.

See Wikipedia article Partition_(number_theory).

EXAMPLES:

```python
sage: from sage.libs.flint.arith import number_of_partitions
sage: number_of_partitions(3)
3
sage: number_of_partitions(10)
42
sage: number_of_partitions(40)
37338
```
```
sage: number_of_partitions(100)
190569292
sage: number_of_partitions(100000)
27493510569775696512677516320986352688173429315980054758203125984302147328114964173055050741660736621590157844774296248940...
4.1 The Elliptic Curve Method for Integer Factorization (ECM)

Sage includes GMP-ECM, which is a highly optimized implementation of Lenstra’s elliptic curve factorization method. See http://ecm.gforge.inria.fr/ for more about GMP-ECM. This file provides a Cython interface to the GMP-ECM library.

AUTHORS:

- Jeroen Demeyer (2012-03-29): signal handling, documentation
- Paul Zimmermann (2011-05-22) – added input/output of sigma

EXAMPLES:

```python
sage: from sage.libs.libecm import ecmfactor
sage: result = ecmfactor(999, 0.00)
sage: result[0]
True
sage: result[1] in [3, 9, 27, 37, 111, 333, 999] or result[1]
True
sage: result = ecmfactor(999, 0.00, verbose=True)
Performing one curve with B1=0
Found factor in step 1: ...
sage: result[0]
True
sage: result[1] in [3, 9, 27, 37, 111, 333, 999] or result[1]
True
sage: ecmfactor(2^128+1,1000,sigma=227140902)
(True, 5704689200685129054721, 227140902)
```

`sage.libs.libecm.ecmfactor(number, B1, verbose=False, sigma=0)`

Try to find a factor of a positive integer using ECM (Elliptic Curve Method). This function tries one elliptic curve.

INPUT:

- `number` – positive integer to be factored
- `B1` – bound for step 1 of ECM
- `verbose` (default: False) – print some debugging information

OUTPUT:

Either `(False, None)` if no factor was found, or `(True, f)` if the factor `f` was found.
EXAMPLES:

```
sage: from sage.libs.libecm import ecmfactor
```

This number has a small factor which is easy to find for ECM:

```
sage: N = 2^167 - 1
sage: factor(N)
   2349023 * 7963830476685650737778616296087448490695649
sage: ecmfactor(N, 2e5)
   (True, 2349023, ...)
```

If a factor was found, we can reproduce the factorization with the same sigma value:

```
sage: N = 2^167 - 1
sage: ecmfactor(N, 2e5, sigma=1473308225)
   (True, 2349023, 1473308225)
```

With a smaller B1 bound, we may or may not succeed:

```
sage: ecmfactor(N, 1e2)  # random
   (False, None)
```

The following number is a Mersenne prime, so we don’t expect to find any factors (there is an extremely small chance that we get the input number back as factorization):

```
sage: N = 2^127 - 1
sage: N.is_prime()
   True
sage: ecmfactor(N, 1e3)
   (False, None)
```

If we have several small prime factors, it is possible to find a product of primes as factor:

```
sage: N = 2^179 - 1
sage: factor(N)
   359 * 1433 * 14894591093003986456940197095433721664951999121
sage: ecmfactor(N, 1e3)  # random
   (True, 514447, 3475102204)
```

We can ask for verbose output:

```
sage: N = 12^97 - 1
sage: factor(N)
   11 * ...
```

```
sage: ecmfactor(N, 100, verbose=True)
   Performing one curve with B1=100
   Found factor in step 1: 11
   (True, 11, ...)
sage: ecmfactor(N/11, 100, verbose=True)
   Performing one curve with B1=100
   Found no factor.
   (False, None)
```
5.1 GSL arrays

```python
class sage.libs.gsl.array.GSLDoubleArray
    Bases: object

    EXAMPLES:

    sage: a = WaveletTransform(128, 'daubechies', 4)
    sage: for i in range(1, 11):
    ....:     a[i] = 1
    sage: a[:6:2]
    [0.0, 1.0, 1.0]
```
6.1 Rubinstein’s lcalc library

This is a wrapper around Michael Rubinstein’s lcalc. See http://oto.math.uwaterloo.ca/~mrubinst/L_function_public/CODE/.

AUTHORS:

• Rishikesh (2010): added compute_rank() and hardy_z_function()
• Yann Laigle-Chapuy (2009): refactored
• Rishikesh (2009): initial version

class sage.libs.lcalc.lcalc_Lfunction.Lfunction
Bases: object

Initialization of L-function objects. See derived class for details, this class is not supposed to be instantiated directly.

EXAMPLES:

```
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: Lfunction_from_character(DirichletGroup(5)[1])
```
L-function with complex Dirichlet coefficients

```
sage: chi=DirichletGroup(5)[2]  #This is a quadratic character
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: L=Lfunction_from_character(chi, type="int")
sage: L.compute_rank()
0
```

```
sage: E=EllipticCurve([-82,0])
sage: L=Lfunction_from_elliptic_curve(E, number_of_coeffs=40000)
sage: L.compute_rank()
3
```

compute_rank()
Computes the analytic rank (the order of vanishing at the center) of of the L-function

EXAMPLES:

```
sage: chi=DirichletGroup(5)[2]  #This is a quadratic character
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: L=Lfunction_from_character(chi, type="int")
sage: L.compute_rank()
0
```

```
sage: E=EllipticCurve([-82,0])
sage: L=Lfunction_from_elliptic_curve(E, number_of_coeffs=40000)
sage: L.compute_rank()
3
```

find_zeros(T1, T2, stepsize)
Finds zeros on critical line between T1 and T2 using step size of stepsize. This function might miss zeros if step size is too large. This function computes the zeros of the L-function by using change in signs of areal valued function whose zeros coincide with the zeros of L-function.

Use find_zeros_via_N() for slower but more rigorous computation.
INPUT:

- T1 – a real number giving the lower bound
- T2 – a real number giving the upper bound
- stepsize – step size to be used for the zero search

OUTPUT:

list – A list of the imaginary parts of the zeros which were found.

EXAMPLES:

```sage
from sage.libs.lcalc.lcalc_Lfunction import *
sage: chi=DirichletGroup(5)[2]  # This is a quadratic character
sage: L=Lfunction_from_character(chi, type="int")
sage: L.find_zeros(5,15,.1)
[6.6485334472..., 9.8314443288..., 11.9588456260...]
sage: L=Lfunction_from_character(chi, type="double")
sage: L.find_zeros(1,15,.1)
[6.6485334472..., 9.8314443288..., 11.9588456260...]
sage: chi=DirichletGroup(5)[1]
sage: L=Lfunction_from_character(chi, type="complex")
sage: L.find_zeros(-8,8,.1)
[-4.13290370521..., 6.18357819545...]
sage: L=Lfunction_Zeta()
sage: L.find_zeros(10,29.1,.1)
[14.1347251417..., 21.0220396387..., 25.0108575801...]
```

`find_zeros_via_N(count=0, do_negative=False, max_refine=1025, rank=-1, test_explicit_formula=0)`

Finds count number of zeros with positive imaginary part starting at real axis. This function also verifies that all the zeros have been found.

INPUT:

- count - number of zeros to be found
- do_negative - (default: False) False to ignore zeros below the real axis.
- max_refine - when some zeros are found to be missing, the step size used to find zeros is refined. max_refine gives an upper limit on when lcalc should give up. Use default value unless you know what you are doing.
- rank - integer (default: -1) analytic rank of the L-function. If -1 is passed, then we attempt to compute it. (Use default if in doubt)
- test_explicit_formula - integer (default: 0) If nonzero, test the explicit formula for additional confidence that all the zeros have been found and are accurate. This is still being tested, so using the default is recommended.

OUTPUT:

list – A list of the imaginary parts of the zeros that have been found

EXAMPLES:
```python
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: chi=DirichletGroup(5)[2]  # This is a quadratic character
sage: L=Lfunction_from_character(chi, type="int")
sage: L.find_zeros_via_N(3)  
[6.64845334472..., 9.83144443288..., 11.9588456260...]
sage: L=Lfunction_from_character(chi, type="double")
sage: L.find_zeros_via_N(3)  
[6.64845334472..., 9.83144443288..., 11.9588456260...]
sage: chi=DirichletGroup(5)[1]
sage: L=Lfunction_from_character(chi, type="complex")
sage: L.find_zeros_via_N(3)  
[6.18357819545..., 8.45722917442..., 12.6749464170...]
sage: L=Lfunction_Zeta()
sage: L.find_zeros_via_N(3)  
[14.1347251417..., 21.0220396387..., 25.0108575801...]
```

**hardy_z_function(s)**
Computes the Hardy Z-function of the L-function at s

**INPUT:**
- s - a complex number with imaginary part between -0.5 and 0.5

**EXAMPLES:**
```python
sage: chi = DirichletGroup(5)[2]  # Quadratic character
sage: from sage.libs.lcalc.lcalc_Lfunction import *
sage: L = Lfunction_from_character(chi, type="int")
sage: L.hardy_z_function(0)
0.231750947504...
sage: L.hardy_z_function(.5).imag()  # abs tol 1e-15
1.17253174178320e-17
sage: L.hardy_z_function(.4+.3*I)
0.2166144222685... - 0.00408187127850...*I
sage: chi = DirichletGroup(5)[1]
sage: L = Lfunction_from_character(chi, type="complex")
sage: L.hardy_z_function(0)
0.793967590477...
sage: L.hardy_z_function(.5).imag()  # abs tol 1e-15
0.000000000000000
sage: E = EllipticCurve([-82,0])
sage: L = Lfunction_from_elliptic_curve(E, number_of_coeffs=40000)
sage: L.hardy_z_function(2.1)
-0.00643179176869...
sage: L.hardy_z_function(2.1).imag()  # abs tol 1e-15
-3.93833660115668e-19
```

**value(s, derivative=0)**
Computes the value of the L-function at s

**INPUT:**
- s - a complex number
- derivative - integer (default: 0) the derivative to be evaluated
- rotate - (default: False) If True, this returns the value of the Hardy Z-function (sometimes called
the Riemann-Siegel Z-function or the Siegel Z-function).

EXAMPLES:

```python
sage: chi=DirichletGroup(5)[2]  # This is a quadratic character
sage: from sage.libs.lcalc.lcalc_Lfunction import *

sage: L=Lfunction_from_character(chi, type="int")

sage: L.value(.5)  # abs tol 3e-15
0.231750947504016 + 5.75329642226136e-18*I

sage: L.value(.2+.4*I)
0.102558603193... + 0.190840777924...*I

sage: L=Lfunction_from_character(chi, type="double")

sage: L.value(.6)  # abs tol 3e-15
0.27463335856345 + 6.59869267328199e-18*I

sage: L.value(.6+I)
0.362258705721... + 0.433888250620...*I

sage: chi=DirichletGroup(5)[1]

sage: L=Lfunction_from_character(chi, type="complex")

sage: L.value(.5)
0.763747880117... + 0.216964767518...*I

sage: L.value(.6+5*I)
0.702723260619... - 1.10178575243...*I

sage: L=Lfunction_Zeta()

sage: L.value(.5)
-1.46035450880...

sage: L.value(.4+.5*I)
-0.450728958517... - 0.780511403019...*I
```

class sage.libs.lcalc.lcalc_Lfunction.Lfunction_C

Bases: sage.libs.lcalc.lcalc_Lfunction.Lfunction

The Lfunction_C class is used to represent L-functions with complex Dirichlet Coefficients. We assume that L-functions satisfy the following functional equation.

\[ \Lambda(s) = \omega Q^s \overline{\Lambda(1-s)} \]

where

\[ \Lambda(s) = Q^s \left( \prod_{j=1}^{\alpha} \Gamma(\kappa_j s + \gamma_j) \right) L(s) \]

See (23) in Arxiv math/0412181

INPUT:

- `what_type_L` - integer, this should be set to 1 if the coefficients are periodic and 0 otherwise.
- `dirichlet_coefficient` - List of dirichlet coefficients of the L-function. Only first \( M \) coefficients are needed if they are periodic.
- `period` - If the coefficients are periodic, this should be the period of the coefficients.
- `Q` - See above
- `OMEGA` - See above
- `kappa` - List of the values of \( \kappa_j \) in the functional equation
- `gamma` - List of the values of \( \gamma_j \) in the functional equation
• pole - List of the poles of L-function
• residue - List of the residues of the L-function

Note: If an L-function satisfies \( \Lambda(s) = \omega Q^s \Lambda(k - s) \), by replacing \( s \) by \( s + (k - 1)/2 \), one can get it in the form we need.

class sage.libs.lcalc.lcalc_Lfunction.Lfunction_D
Bases: sage.libs.lcalc.lcalc_Lfunction.Lfunction

The Lfunction_D class is used to represent L-functions with real Dirichlet coefficients. We assume that L-functions satisfy the following functional equation.

\[
\Lambda(s) = \omega Q^s \Lambda(1 - s)
\]

where

\[
\Lambda(s) = Q^s \left( \prod_{j=1}^{a} \Gamma(\kappa_j s + \gamma_j) \right) L(s)
\]

See (23) in Arxiv math/0412181

INPUT:
• what_type_L - integer, this should be set to 1 if the coefficients are periodic and 0 otherwise.
• dirichlet_coefficient - List of dirichlet coefficients of the L-function. Only first \( M \) coefficients are needed if they are periodic.
• period - If the coefficients are periodic, this should be the period of the coefficients.
• Q - See above
• OMEGA - See above
• kappa - List of the values of \( \kappa_j \) in the functional equation
• gamma - List of the values of \( \gamma_j \) in the functional equation
• pole - List of the poles of L-function
• residue - List of the residues of the L-function

Note: If an L-function satisfies \( \Lambda(s) = \omega Q^s \Lambda(k - s) \), by replacing \( s \) by \( s + (k - 1)/2 \), one can get it in the form we need.

class sage.libs.lcalc.lcalc_Lfunction.Lfunction_I
Bases: sage.libs.lcalc.lcalc_Lfunction.Lfunction

The Lfunction_I class is used to represent L-functions with integer Dirichlet Coefficients. We assume that L-functions satisfy the following functional equation.

\[
\Lambda(s) = \omega Q^s \Lambda(1 - s)
\]

where

\[
\Lambda(s) = Q^s \left( \prod_{j=1}^{a} \Gamma(\kappa_j s + \gamma_j) \right) L(s)
\]
See (23) in Arxiv math/0412181

INPUT:
- `what_type_L` - integer, this should be set to 1 if the coefficients are periodic and 0 otherwise.
- `dirichlet_coefficient` - List of dirichlet coefficients of the L-function. Only first $M$ coefficients are needed if they are periodic.
- `period` - If the coefficients are periodic, this should be the period of the coefficients.
- `Q` - See above
- `OMEGA` - See above
- `kappa` - List of the values of $\kappa_j$ in the functional equation
- `gamma` - List of the values of $\gamma_j$ in the functional equation
- `pole` - List of the poles of L-function
- `residue` - List of the residues of the L-function

Note: If an L-function satisfies $\Lambda(s) = \omega Q s \Lambda(k - s)$, by replacing $s$ by $s + (k - 1)/2$, one can get it in the form we need.

class sage.libs.lcalc.lcalc_Lfunction.Lfunction_Zeta
Bases: sage.libs.lcalc.lcalc_Lfunction.Lfunction

The Lfunction_Zeta class is used to generate the Riemann zeta function.

sage.libs.lcalc.lcalc_Lfunction.Lfunction_from_character (chi, type='complex')

Given a primitive Dirichlet character, this function returns an lcalc L-function object for the L-function of the character.

INPUT:
- `chi` - A Dirichlet character
- `use_type` - string (default: “complex”) type used for the Dirichlet coefficients. This can be “int”, “double” or “complex”.

OUTPUT:
L-function object for `chi`.

EXAMPLES:

L-function object for `chi`.

```python
sage: from sage.libs.lcalc.lcalc_Lfunction import Lfunction_from_character
sage: Lfunction_from_character(DirichletGroup(5)[1])
L-function with complex Dirichlet coefficients
sage: Lfunction_from_character(DirichletGroup(5)[2], type="int")
L-function with integer Dirichlet coefficients
sage: Lfunction_from_character(DirichletGroup(5)[2], type="double")
L-function with real Dirichlet coefficients
sage: Lfunction_from_character(DirichletGroup(5)[1], type="int")
Traceback (most recent call last):
  ...
ValueError: For non quadratic characters you must use type="complex"
```

sage.libs.lcalc.lcalc_Lfunction.Lfunction_from_elliptic_curve (E, number_of_coeffs=10000)

Given an elliptic curve E, return an L-function object for the function $L(s, E)$.
INPUT:

- E - An elliptic curve
- number_of_coeffs - integer (default: 10000) The number of coefficients to be used when constructing the L-function object. Right now this is fixed at object creation time, and is not automatically set intelligently.

OUTPUT:

L-function object for $L(s, E)$.

EXAMPLES:

```python
sage: from sage.libs.lcalc.lcalc_Lfunction import Lfunction_from_elliptic_curve
sage: L = Lfunction_from_elliptic_curve(EllipticCurve('37'))
sage: L
L-function with real Dirichlet coefficients
sage: L.value(0.5).abs() < 1e-15  # "noisy" zero on some platforms (see #9615)
True
sage: L.value(0.5, derivative=1)
0.305999...
```
7.1 libSingular: Functions

Sage implements a C wrapper around the Singular interpreter which allows to call any function directly from Sage without string parsing or interprocess communication overhead. Users who do not want to call Singular functions directly, usually do not have to worry about this interface, since it is handled by higher level functions in Sage.

AUTHORS:

• Michael Brickenstein (2009-07): initial implementation, overall design
• Martin Albrecht (2009-07): clean up, enhancements, etc.
• Michael Brickenstein (2009-10): extension to more Singular types
• Martin Albrecht (2010-01): clean up, support for attributes
• Simon King (2011-04): include the documentation provided by Singular as a code block.
• Burcin Erocal, Michael Brickenstein, Oleksandr Motsak, Alexander Dreyer, Simon King (2011-09) plural support

EXAMPLES:

The direct approach for loading a Singular function is to call the function `singular_function()` with the function name as parameter:

```python
sage: from sage.libs.singular.function import singular_function
sage: P.<a,b,c,d> = PolynomialRing(GF(7))

sage: std = singular_function('std')

sage: I = sage.rings.ideal.Cyclic(P)

sage: std(I)
[a + b + c + d,
b^2 + 2*b*d + d^2,
b*c^2 + c^2*d - b*d^2 - d^3,
b*c*d^2 + c^2*d^2 - b*d^3 + c*d^3 - d^4 - 1,
b*d^4 + d^5 - b - d,
c^3*d^2 + c^2*d^3 - c - d,
c^2*d^4 + b*c - b*d + c*d - 2*d^2]
```

If a Singular library needs to be loaded before a certain function is available, use the `lib()` function as shown below:

```python
sage: from sage.libs.singular.function import singular_function, lib as singular_lib

sage: primdecSY = singular_function('primdecSY')

Traceback (most recent call last):
...
NameError: Singular library function 'primdecSY' is not defined
```

(continues on next page)
There is also a short-hand notation for the above:

```
sage: import sage.libs.singular.function_factory
sage: primdecSY = sage.libs.singular.function_factory.ff.primdec__lib.primdecSY
```

The above line will load “primdec.lib” first and then load the function primdecSY.

---

**Definition:**

- **sage.libs.singular.function.BaseCallHandler**
  ```python
  class sage.libs.singular.function.BaseCallHandler
  Bases: object
  A call handler is an abstraction which hides the details of the implementation differences between kernel and library functions.
  ```

- **sage.libs.singular.function.Converter**
  ```python
  class sage.libs.singular.function.Converter
  Bases: sage.structure.sage_object.SageObject
  A Converter interfaces between Sage objects and Singular interpreter objects.
  ```

  ```python
  ring()
  Return the ring in which the arguments of this list live.
  ```

  **Examples:**

  ```python
  sage: from sage.libs.singular.function import Converter
  sage: P.<a,b,c> = PolynomialRing(GF(127))
  sage: Converter([a,b,c],ring=P).ring()
  Multivariate Polynomial Ring in a, b, c over Finite Field of size 127
  ```

- **sage.libs.singular.function.KernelCallHandler**
  ```python
  class sage.libs.singular.function.KernelCallHandler
  Bases: sage.libs.singular.function.BaseCallHandler
  A call handler is an abstraction which hides the details of the implementation differences between kernel and library functions.
  ```

  **Note:** Do not construct this class directly, use `singular_function()` instead.

- **sage.libs.singular.function.LibraryCallHandler**
  ```python
  class sage.libs.singular.function.LibraryCallHandler
  Bases: sage.libs.singular.function.BaseCallHandler
  A call handler is an abstraction which hides the details of the implementation differences between kernel and library functions.
  ```

  **Note:** Do not construct this class directly, use `singular_function()` instead.

- **sage.libs.singular.function.Resolution**
  ```python
  class sage.libs.singular.function.Resolution
  Bases: object
  A simple wrapper around Singular’s resolutions.
  ```
class sage.libs.singular.function.RingWrap
Bases: object

A simple wrapper around Singular’s rings.

characteristic()
Get characteristic.

EXAMPLES:

```
sage: from sage.libs.singular.function import singular_function
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: ringlist = singular_function("ringlist")
sage: l = ringlist(P)
sage: ring = singular_function("ring")
sage: ring(l, ring=P).characteristic()
sage: 0
```

is_commutative()
Determine whether a given ring is commutative.

EXAMPLES:

```
sage: from sage.libs.singular.function import singular_function
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: ringlist = singular_function("ringlist")
sage: l = ringlist(P)
sage: ring = singular_function("ring")
sage: ring(l, ring=P).is_commutative()
sage: True
```

ngens()
Get number of generators.

EXAMPLES:

```
sage: from sage.libs.singular.function import singular_function
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: ringlist = singular_function("ringlist")
sage: l = ringlist(P)
sage: ring = singular_function("ring")
sage: ring(l, ring=P).ngens()
sage: 3
```

npars()
Get number of parameters.

EXAMPLES:

```
sage: from sage.libs.singular.function import singular_function
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: ringlist = singular_function("ringlist")
sage: l = ringlist(P)
sage: ring = singular_function("ring")
sage: ring(l, ring=P).npars()
sage: 0
```

ordering_string()
Get Singular string defining monomial ordering.

EXAMPLES:
par_names() Get parameter names.

```python
par_names()

Get parameter names.

EXAMPLES:

```import from sage.libs.singular.function import singular_function
P.<x,y,z> = PolynomialRing(QQ)
ringlist = singular_function("ringlist")
l = ringlist(P)
ring = singular_function("ring")
rings = ring(l, ring=P).ordering_string()
rings
'
```

var_names() Get names of variables.

```python
var_names()

Get names of variables.

EXAMPLES:

```import from sage.libs.singular.function import singular_function
P.<x,y,z> = PolynomialRing(QQ)
ringlist = singular_function("ringlist")
l = ringlist(P)
ring = singular_function("ring")
rings = ring(l, ring=P).par_names()
rings
['x', 'y', 'z']
```

class sage.libs.singular.function.SingularFunction

Bases: sage.structure.sage_object.SageObject

The base class for Singular functions either from the kernel or from the library.

class sage.libs.singular.function.SingularKernelFunction

Bases: sage.libs.singular.function.SingularFunction

EXAMPLES:

```python
R.<x,y> = PolynomialRing(QQ, order='lex')
I = R.ideal(x, x+1)
f = SingularKernelFunction("std")
f(I)
[1]
```

class sage.libs.singular.function.SingularLibraryFunction

Bases: sage.libs.singular.function.SingularFunction

EXAMPLES:

```python
R.<x,y> = PolynomialRing(QQ, order='lex')
I = R.ideal(x, x+1)
```
sage: f = SingularLibraryFunction("groebner")
sage: f(I)
[1]

sage.libs.singular.function.all_singular_poly_wrapper(s)
Tests for a sequence s, whether it consists of singular polynomials.

EXAMPLES:

sage: from sage.libs.singular.function import all_singular_poly_wrapper
sage: P.<x,y,z> = QQ[]
sage: all_singular_poly_wrapper([x+1, y])
True
sage: all_singular_poly_wrapper([x+1, y, 1])
False

sage.libs.singular.function.all_vectors(s)
Checks if a sequence s consists of free module elements over a singular ring.

EXAMPLES:

sage: from sage.libs.singular.function import all_vectors
sage: P.<x,y,z> = QQ[]
sage: M = P**2
sage: all_vectors([x])
False
sage: all_vectors([(x,y)])
False
sage: all_vectors([M(0), M((x,y))])
True
sage: all_vectors([M(0), M((x,y)),(0,0)])
False

sage.libs.singular.function.is_sage_wrapper_for_singular_ring(ring)
Check whether wrapped ring arises from Singular or Singular/Plural.

EXAMPLES:

sage: from sage.libs.singular.function import is_sage_wrapper_for_singular_ring
sage: P.<x,y,z> = QQ[]
sage: is_sage_wrapper_for_singular_ring(P)
True
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: P = A.g_algebra(relations={y*x:-x*y}, order = 'lex')
sage: is_sage_wrapper_for_singular_ring(P)
True

sage.libs.singular.function.is_singular_poly_wrapper(p)
Checks if p is some data type corresponding to some singular poly.

EXAMPLES:

sage: from sage.libs.singular.function import is_singular_poly_wrapper
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({z*x:x*z+2*x, z*y:y*z-2*y})
sage: is_singular_poly_wrapper(x*y)
True
sage.libs.singular.function.lib(name)

Load the Singular library name.

INPUT:

• name – a Singular library name

EXAMPLES:

```python
sage: from sage.libs.singular.function import singular_function
sage: from sage.libs.singular.function import lib as singular_lib
sage: singular_lib('general.lib')
```

sage.libs.singular.function.list_of_functions(packages=False)

Return a list of all function names currently available.

INPUT:

• packages – include local functions in packages.

EXAMPLES:

```python
sage: from sage.libs.singular.function import list_of_functions
sage: 'groebner' in list_of_functions()
True
```

sage.libs.singular.function.singular_function(name)

Construct a new libSingular function object for the given name.

This function works both for interpreter and built-in functions.

INPUT:

• name – the name of the function

EXAMPLES:

```python
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: f = 3*x*y + 2*z + 1
sage: g = 2*x + 1/2
sage: I = Ideal([f,g])
sage: from sage.libs.singular.function import singular_function
sage: std = singular_function("std")
sage: std(I)
[3*y - 8*z - 4, 4*x + 1]
sage: size = singular_function("size")
sage: size([2, 3, 3])
3
sage: size("sage")
4
sage: size(["hello", "sage"])
2
sage: factorize = singular_function("factorize")
sage: factorize(f)
[[1, 3*x*y + 2*z + 1], (1, 1)]
sage: factorize(f, 1)
[3*x*y + 2*z + 1]
```
We give a wrong number of arguments:

```python
sage: factorize()
Traceback (most recent call last):
...  
RuntimeError: error in Singular function call 'factorize':
Wrong number of arguments (got 0 arguments, arity code is 303)
sage: factorize(f, 1, 2)
Traceback (most recent call last):
...  
RuntimeError: error in Singular function call 'factorize':
Wrong number of arguments (got 3 arguments, arity code is 303)
sage: factorize(f, 1, 2, 3)
Traceback (most recent call last):
...  
RuntimeError: error in Singular function call 'factorize':
Wrong number of arguments (got 4 arguments, arity code is 303)
```

The Singular function `list` can be called with any number of arguments:

```python
sage: singular_list = singular_function("list")
sage: singular_list(2, 3, 6)
[2, 3, 6]
sage: singular_list()  
[]
sage: singular_list(1)  
[1]
sage: singular_list(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We try to define a non-existing function:

```python
sage: number_foobar = singular_function('number_foobar')
Traceback (most recent call last):
...  
NameError: Singular library function 'number_foobar' is not defined
```

```python
sage: from sage.libs.singular.function import lib as singular_lib
sage: singular_lib('general.lib')
sage: number_e = singular_function('number_e')
sage: number_e(10r)  
67957045707/25000000000
sage: RR(number_e(10r))  
2.71828182828000
```

```python
sage: singular_lib('primdec.lib')
sage: primdecGTZ = singular_function("primdecGTZ")
sage: primdecGTZ(I)  
[[[y - 8/3*z - 4/3, x + 1/4], [y - 8/3*z - 4/3, x + 1/4]]]
sage: singular_list({1,2,3,3,[1,2,3]}, ring=P)  
[(1, 2, 3), 3, [1, 2, 3]]
sage: ringlist=singular_function("ringlist")
sage: 1 = ringlist(P)
sage: 1[3].__class__
<class 'sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic'>
sage: 1
```

(continues on next page)
\[0, ['x', 'y', 'z'], [[dp', (1, 1, 1), ['C', (0,)]], [0]]\]
sage: ring=singular_function("ring")
sage: ring(1)
<RingWrap>
sage: matrix = Matrix(P, 2, 2)
sage: matrix.randomize(terms=1)
sage: det = singular_function("det")
sage: det(matrix)
-3/5*x*y*z
sage: coeffs = singular_function("coeffs")
sage: coeffs(x*y+y+1,y)
[1]
[x + 1]
sage: intmat = Matrix(ZZ, 2, 2, [100, 2, 3, 4])
sage: det(intmat)
394
sage: random = singular_function("random")
sage: A = random(10, 2, 3); A.nrows(), max(A.list()) <= 10
(2, True)
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: M=P**3
sage: leadcoef = singular_function("leadcoef")
sage: lead(v)
(0, y^3)
sage: jet = singular_function("jet")
sage: jet(v, 2)
(x + y, x*y, z)
sage: syz = singular_function("syz")
sage: I = P.ideal([x*y, x+y-x, x**2+1])
sage: M = syz(I)
sage: M
[(-2*y, 2, y + 1, 0), (0, -2, x - 1, 0), (x*y - y, -y + 1, 1, -y), (x^2 + 1, -x - 1, -1, -x)]
sage: singular_lib("mprimdec.lib")
sage: syz(M)
[(-x - 1, y - 1, 2*x, -2*y)]
sage: GTZmod = singular_function("GTZmod")
sage: GTZmod(M)

sage: resolution = mres(M, 0)
<Resolution>
sage: singular_list(resolution)

sage: A.<x,y> = FreeAlgebra(QQ, 2)
sage: P.<x,y> = A.g_algebra({y*x:-x*y})
sage: I= Sequence([x*y, x+y], check=False, immutable=True)
sage: twostd = singular_function("twostd")

(continues on next page)
sage: twostd(I)
[x + y, y^2]
sage: M=syz(I)
doctest...
sage: M
[[(x + y, x*y)]
sage: syz(M)
[(0)]
sage: mres(I, 0)
<Resolution>
sage: M=P**3
sage: v=M((100*x,5*y,10*y*x*y))
sage: leadcoef(v)
-10
sage: v = M([x+y,x*y+y**3,x])
sage: lead(v)
(0, y^3)
sage: jet(v, 2)
(x + y, x*y, x)
sage: l = ringlist(P)
sage: len(l)
6
sage: ring(l)
<noncommutative RingWrap>
sage: I=twostd(I)
sage: I[3]=I
sage: ring(l)
<noncommutative RingWrap>

7.2 libSingular: Function Factory

AUTHORS:

- Martin Albrecht (2010-01): initial version

class sage.libs.singular.function_factory.SingularFunctionFactory
Bases: object

A convenient interface to libsingular functions.

trait_names()  # EXAMPLES:

sage: import sage.libs.singular.function_factory
sage: "groebner" in sage.libs.singular.function_factory.ff.trait_names()
True

7.3 libSingular: Conversion Routines and Initialisation

AUTHOR:

- Martin Albrecht <malb@informatik.uni-bremen.de>
7.4 Wrapper for Singular’s Polynomial Arithmetic

AUTHOR:

- Martin Albrecht (2009-07): refactoring

7.5 libSingular: Options

Singular uses a set of global options to determine verbosity and the behavior of certain algorithms. We provide an interface to these options in the most ‘natural’ python-ic way. Users who do not wish to deal with Singular functions directly usually do not have to worry about this interface or Singular options in general since this is taken care of by higher level functions.

We compute a Groebner basis for Cyclic-5 in two different contexts:

```
sage: P.<a,b,c,d,e> = PolynomialRing(GF(127))
sage: I = sage.rings.ideal.Cyclic(P)
sage: import sage.libs.singular.function_factory
sage: std = sage.libs.singular.function_factory.ff.std
```

By default, tail reductions are performed:

```
sage: from sage.libs.singular.option import opt, opt_ctx
sage: opt['red_tail']
True
sage: std(I)[-1]
d^2*e^6 + 28*b*c*d + ...
```

If we don’t want this, we can create an option context, which disables this:

```
sage: with opt_ctx(red_tail=False, red_sb=False):
....: std(I)[-1]
d^2*e^6 + 8*c^3 + ...
```

However, this does not affect the global state:

```
sage: opt['red_tail']
True
```

On the other hand, any assignment to an option object will immediately change the global state:

```
sage: opt['red_tail'] = False
sage: opt['red_tail']
False
sage: opt['red_tail'] = True
sage: opt['red_tail']
True
```

Assigning values within an option context, only affects this context:

```
sage: with opt_ctx:
....: opt['red_tail'] = False
sage: opt['red_tail']
True
```
Option contexts can also be safely stacked:

```python
sage: with opt_ctx:
    ....: opt['red_tail'] = False
    ....: print(opt)
    ....: with opt_ctx:
    ....:     opt['red_through'] = False
    ....:     print(opt)
```

```
general options for libSingular (current value 0x00000082)
general options for libSingular (current value 0x00000002)
```

```python
sage: print(opt)
```

```
general options for libSingular (current value 0x02000082)
```

Furthermore, the integer valued options `deg_bound` and `mult_bound` can be used:

```python
sage: R.<x,y> = QQ[]
sage: I = R*[x^3+y^2,x^2*y+1]
sage: opt['deg_bound'] = 2
sage: std(I)  
[x^2*y + 1, x^3 + y^2]
sage: opt['deg_bound'] = 0
sage: std(I)  
[y^3 - x, x^2*y + 1, x^3 + y^2]
```

The same interface is available for verbosity options:

```python
sage: from sage.libs.singular.option import opt_verb
sage: opt_verb['not_warn_sb']
False
sage: opt.reset_default()  # needed to avoid side effects
sage: opt_verb.reset_default()  # needed to avoid side effects
```

**AUTHOR:**

- Martin Albrecht (2009-08): initial implementation
- Martin Albrecht (2010-01): better interface, verbosity options
- Simon King (2010-07): Python-ic option names; `deg_bound` and `mult_bound`

**class** `sage.libs.singular.option.LibSingularOptions`  
Bases: `sage.libs.singular.option.LibSingularOptions_abstract`  

Pythonic Interface to libSingular’s options.

Supported options are:

- `return_sb` or `returnSB` - the functions `syz, intersect, quotient, modulo` return a standard base instead of a generating set if `return_sb` is set. This option should not be used for `lift`.
- `fast_hc` or `fastHC` - tries to find the highest corner of the staircase (HC) as fast as possible during a standard basis computation (only used for local orderings).
- `int_strategy` or `intStrategy` - avoids division of coefficients during standard basis computations. This option is ring dependent. By default, it is set for rings with characteristic 0 and not set for all other rings.
- `lazy` - uses a more lazy approach in `std` computations, which was used in SINGULAR version before 2-0 (and which may lead to faster or slower computations, depending on the example).
- `length` - select shorter reducers in `std` computations.
• `not_regularity` or `notRegulararity` - disables the regularity bound for `res` and `mres`.

• `not_sugar` or `notSugar` - disables the sugar strategy during standard basis computation.

• `not_buckets` or `notBuckets` - disables the bucket representation of polynomials during standard basis computations. This option usually decreases the memory usage but increases the computation time. It should only be set for memory-critical standard basis computations.

• `old_std` or `oldStd` - uses a more lazy approach in std computations, which was used in SINGULAR version before 2-0 (and which may lead to faster or slower computations, depending on the example).

• `prot` - shows protocol information indicating the progress during the following computations: `facstd`, `fglm`, `groebner`, `lres`, `mres`, `minres`, `mstd`, `res`, `slimgb`, `sres`, `std`, `stdfglm`, `stdhilb`, `syz`.

• `red,b` or `redSB` - computes a reduced standard basis in any standard basis computation.

• `red_tail` or `redTail` - reduction of the tails of polynomials during standard basis computations. This option is ring dependent. By default, it is set for rings with global degree orderings and not set for all other rings.

• `red_through` or `redThrough` - for inhomogenous input, polynomial reductions during standard basis computations are never postponed, but always finished through. This option is ring dependent. By default, it is set for rings with global degree orderings and not set for all other rings.

• `sugar_crit` or `sugarCrit` - uses criteria similar to the homogeneous case to keep more useless pairs.

• `weight_m` or `weightM` - automatically computes suitable weights for the weighted ecart and the weighted sugar method.

In addition, two integer valued parameters are supported, namely:

• `deg_bound` or `degBound` - The standard basis computation is stopped if the total (weighted) degree exceeds `deg_bound`. `deg_bound` should not be used for a global ordering with inhomogeneous input. Reset this bound by setting `deg_bound` to 0. The exact meaning of “degree” depends on the ring ordering and the command: `slimgb` uses always the total degree with weights 1, `std` does so for block orderings, only.

• `mult_bound` or `multBound` - The standard basis computation is stopped if the ideal is zero-dimensional in a ring with local ordering and its multiplicity is lower than `mult_bound`. Reset this bound by setting `mult_bound` to 0.

**EXAMPLES:**

```python
sage: from sage.libs.singular.option import LibSingularOptions
sage: libsingular_options = LibSingularOptions()
```

Here we demonstrate the intended way of using libSingular options:

```python
sage: R.<x,y> = QQ[]
sage: I = R*[x^3+y^2,x^2*y+1]
sage: I.groebner_basis(deg_bound=2)
[x^3 + y^2, x^2*y + 1]
sage: I.groebner_basis()
[x^3 + y^2, x^2*y + 1, y^3 - x]
```

The option `mult_bound` is only relevant in the local case:
```
sage: from sage.libs.singular.option import opt
sage: Rlocal.<x,y,z> = PolynomialRing(QQ, order='ds')
sage: x^2<x
True
sage: J = [x^7+y^7+z^6,x^6+y^8+z^7,x^7+y^5+z^8, x^2*y^3+y^2*z^3+x^3+z^2,x^3+y^2+y^→3*z^2+x^2*z^3]*Rlocal
sage: J.groebner_basis(mult_bound=100)
[x^3*y^2 + y^3*z^2 + x^2*z^3, x^2*y^3 + x^3*z^2 + y^2*z^3, y^5, x^6 + x*y^4*z^5, →x^4*z^2 - y^4*z^2 - x^2*y*z^3 + x*y^2*z^3, z^6 - x*y^4*z^4 - x^3*y*z^5]
sage: opt['red_tail'] = True  # the previous commands reset opt['red_tail'] to False
sage: J.groebner_basis()
[x^3*y^2 + y^3*z^2 + x^2*z^3, x^2*y^3 + x^3*z^2 + y^2*z^3, y^5, x^6, x^4*z^2 - y^→4*z^2 - x^2*y*z^3 + x*y^2*z^3, z^6 - x*y^4*z^4 - x^3*y*z^5]```

**reset_default()**
Reset libSingular's default options.

**EXAMPLES:**
```
sage: from sage.libs.singular.option import opt
sage: opt['red_tail']
True
sage: opt['red_tail'] = False
sage: opt['red_tail']
False
sage: opt['deg_bound']
0
sage: opt['deg_bound'] = 2
sage: opt['deg_bound']
2
sage: opt.reset_default()
sage: opt['red_tail']
True
sage: opt['deg_bound']
0
```

**class sage.libs.singular.option.LibSingularOptionsContext**
Bases: object

Option context
This object localizes changes to options.

**EXAMPLES:**
```
sage: from sage.libs.singular.option import opt, opt_ctx
sage: opt
general options for libSingular (current value 0x06000082)

sage: with opt_ctx(redTail=False):
....: print(opt)
....: with opt_ctx(redThrough=False):
....: print(opt)
go to Singular
```

(continues on next page)
sage: print(opt)
general options for libSingular (current value 0x06000082)

opt
class sage.libs.singular.option.LibSingularOptions_abstract
Bases: object

Abstract Base Class for libSingular options.

load(value=None)
EXAMPLES:

sage: from sage.libs.singular.option import opt as sopt
sage: bck = sopt.save(); hex(bck[0]), bck[1], bck[2]
('0x6000082', 0, 0)
sage: sopt['redTail'] = False
sage: hex(int(sopt))
'0x4000082'
sage: sopt.load(bck)
sage: sopt['redTail']
True

save()
Return a triple of integers that allow reconstruction of the options.

EXAMPLES:

sage: from sage.libs.singular.option import opt
sage: opt['deg_bound']
0
sage: opt['red_tail']
True
sage: s = opt.save()
sage: opt['deg_bound'] = 2
sage: opt['red_tail'] = False
sage: opt['deg_bound']
2
sage: opt['red_tail']
False
sage: opt.load(s)
sage: opt['deg_bound']
0
sage: opt['red_tail']
True
sage: opt.reset_default() # needed to avoid side effects

class sage.libs.singular.option.LibSingularVerboseOptions
Bases: sage.libs.singular.option.LibSingularOptions_abstract

Pytonic Interface to libSingular’s verbosity options.

Supported options are:

- **mem** - shows memory usage in square brackets.
- **yacc** - Only available in debug version.
- **redefine** - warns about variable redefinitions.
- **reading** - shows the number of characters read from a file.
• loadLib or load_lib - shows loading of libraries.
• debugLib or debug_lib - warns about syntax errors when loading a library.
• loadProc or load_proc - shows loading of procedures from libraries.
• defRes or def_res - shows the names of the syzygy modules while converting resolution to list.
• usage - shows correct usage in error messages.
• Imap or imap - shows the mapping of variables with the fetch and imap commands.
• notWarnSB or not_warn_sb - do not warn if a basis is not a standard basis
• contentSB or content_sb - avoids to divide by the content of a polynomial in std and related algorithms. Should usually not be used.
• cancelunit - avoids to divide polynomials by non-constant units in std in the local case. Should usually not be used.

EXAMPLES:

```
sage: from sage.libs.singular.option import LibSingularVerboseOptions
sage: libsingular_verbose = LibSingularVerboseOptions()
```

reset_default()
Return to libSingular's default verbosity options

EXAMPLES:

```
sage: from sage.libs.singular.option import opt_verb
sage: opt_verb['not_warn_sb']
False
sage: opt_verb['not_warn_sb'] = True
sage: opt_verb['not_warn_sb']
True
sage: opt_verb.reset_default()
```

7.6 Wrapper for Singular’s Rings

AUTHORS:

• Martin Albrecht (2009-07): initial implementation
• Kwankyu Lee (2010-06): added matrix term order support

```
sage.libs.singular.ring.currRing_wrapper()
```

Returns a wrapper for the current ring, for use in debugging ring_refcount_dict.

EXAMPLES:

```
sage: from sage.libs.singular.ring import currRing_wrapper
sage: currRing_wrapper()
```

The ring pointer ...
sage.libs.singular.ring.poison_currRing(frame, event, arg)
Poison the currRing pointer.
This function sets the currRing to an illegal value. By setting it as the python debug hook, you can poison the currRing before every evaluated Python command (but not within Cython code).

INPUT:
• frame, event, arg – the standard arguments for the CPython debugger hook. They are not used.

OUTPUT:
Returns itself, which ensures that poison_currRing() will stay in the debugger hook.

EXAMPLES:

```python
sage: previous_trace_func = sys.gettrace()  # None if no debugger running
sage: from sage.libs.singular.ring import poison_currRing
sage: sys.settrace(poison_currRing)
sage: sys.gettrace()
<built-in function poison_currRing>
sage: sys.settrace(previous_trace_func)  # switch it off again
```

sage.libs.singular.ring.print_currRing()
Print the currRing pointer.

EXAMPLES:

```python
sage: from sage.libs.singular.ring import print_currRing
sage: print_currRing()  # random output
DEBUG: currRing == 0x7fc6fa6ec480
sage: from sage.libs.singular.ring import poison_currRing
sage: _ = poison_currRing(None, None, None)
sage: print_currRing()
DEBUG: currRing == 0x0
```

class sage.libs.singular.ring.ring_wrapper_Py
Bases: object
Python object wrapping the ring pointer.
This is useful to store ring pointers in Python containers.
You must not construct instances of this class yourself, use wrap_ring() instead.

EXAMPLES:

```python
sage: from sage.libs.singular.ring import ring_wrapper_Py
sage: ring_wrapper_Py
<type 'sage.libs.singular.ring.ring_wrapper_Py'>
```

### 7.7 Singular’s Groebner Strategy Objects

AUTHORS:
• Martin Albrecht (2009-07): initial implementation
• Michael Brickenstein (2009-07): initial implementation
• Hans Schoenemann (2009-07): initial implementation
class sage.libs.singular.groebner_strategy.GroebnerStrategy
Bases: sage.structure.sage_object.SageObject


This object provides functions for normal form computations and other functions for Groebner basis computation.

ALGORITHM:
Uses Singular via libSINGULAR

ideal()
Return the ideal this strategy object is defined for.

EXAMPLES:

    sage: from sage.libs.singular.groebner_strategy import GroebnerStrategy
    sage: P.<x,y,z> = PolynomialRing(GF(32003))
    sage: I = Ideal([x + z, y + z])
    sage: strat = GroebnerStrategy(I)
    sage: strat.ideal()
    Ideal (x + z, y + z) of Multivariate Polynomial Ring in x, y, z over Finite Field of size 32003

normal_form(p)
Compute the normal form of \( p \) with respect to the generators of this object.

EXAMPLES:

    sage: from sage.libs.singular.groebner_strategy import GroebnerStrategy
    sage: P.<x,y,z> = PolynomialRing(QQ)
    sage: I = Ideal([x + z, y + z])
    sage: strat = GroebnerStrategy(I)
    sage: strat.normal_form(x*y)
    z^2
    sage: strat.normal_form(x + 1)
    -z + 1

ring()
Return the ring this strategy object is defined over.

EXAMPLES:

    sage: from sage.libs.singular.groebner_strategy import GroebnerStrategy
    sage: P.<x,y,z> = PolynomialRing(GF(32003))
    sage: I = Ideal([x + z, y + z])
    sage: strat = GroebnerStrategy(I)
    sage: strat.ring()
    Multivariate Polynomial Ring in x, y, z over Finite Field of size 32003

class sage.libs.singular.groebner_strategy.NCGroebnerStrategy
Bases: sage.structure.sage_object.SageObject


This object provides functions for normal form computations and other functions for Groebner basis computation.

ALGORITHM:
Uses Singular via libSINGULAR

7.7. Singular's Groebner Strategy Objects
ideal()

Return the ideal this strategy object is defined for.

EXAMPLES:

```python
sage: from sage.libs.singular.groebner_strategy import NCGroebnerStrategy
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: I = H.ideal([y^2, x^2, z^2-H.one()])
sage: strat = NCGroebnerStrategy(I)
sage: strat.ideal() == I
True
```

normal_form(p)

Compute the normal form of \( p \) with respect to the generators of this object.

EXAMPLES:

```python
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: JL = H.ideal([x^3, y^3, z^3 - 4*z])
sage: JT = H.ideal([x^3, y^3, z^3 - 4*z], side='twosided')
sage: from sage.libs.singular.groebner_strategy import NCGroebnerStrategy
sage: SL = NCGroebnerStrategy(JL.std())
sage: ST = NCGroebnerStrategy(JT.std())
sage: SL.normal_form(x*y^2)
x*y^2
sage: ST.normal_form(x*y^2)
y*z
```

ring()

Return the ring this strategy object is defined over.

EXAMPLES:

```python
sage: from sage.libs.singular.groebner_strategy import NCGroebnerStrategy
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: I = H.ideal([y^2, x^2, z^2-H.one()])
sage: strat = NCGroebnerStrategy(I)
sage: strat.ring() is H
True
```

sage.libs.singular.groebner_strategy.unpickle_GroebnerStrategy0(I)

EXAMPLES:

```python
sage: from sage.libs.singular.groebner_strategy import GroebnerStrategy
sage: P.<x,y,z> = PolynomialRing(GF(32003))
sage: I = Ideal([x + z, y + z])
sage: strat = GroebnerStrategy(I)
sage: loads(dumps(strat)) == strat # indirect doctest
True
```

sage.libs.singular.groebner_strategy.unpickle_NCGroebnerStrategy0(I)

EXAMPLES:

```python
sage: from sage.libs.singular.groebner_strategy import NCGroebnerStrategy
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: loads(dumps(NCGroebnerStrategy(I))) == NCGroebnerStrategy(I)
True
```
sage: H.<x,y,z> = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: I = H.ideal([y^2, x^2, z^2-H.one()])
sage: strat = NCGroebnerStrategy(I)
sage: loads(dumps(strat)) == strat  # indirect doctest
True
8.1 Context Managers for LibGAP

This module implements a context manager for global variables. This is useful since the behavior of GAP is sometimes controlled by global variables, which you might want to switch to a different value for a computation. Here is an example how you are suppose to use it from your code. First, let us set a dummy global variable for our example:

```
sage: libgap.set_global('FooBar', 123)
```

Then, if you want to switch the value momentarily you can write:

```
sage: with libgap.global_context('FooBar', 'test'):
    ....: print(libgap.get_global('FooBar'))
```

Afterward, the global variable reverts to the previous value:

```
sage: print(libgap.get_global('FooBar'))
123
```

The value is reset even if exceptions occur:

```
sage: with libgap.global_context('FooBar', 'test'):
    ....: print(libgap.get_global('FooBar'))
    ....: raise ValueError(libgap.get_global('FooBar'))
```

```
Traceback (most recent call last):
  ...
ValueError: test
```

```
sage: print(libgap.get_global('FooBar'))
123
```

class sage.libs.gap.context_managers.GlobalVariableContext (variable, value)
Context manager for GAP global variables.

It is recommended that you use the `sage.libs.gap.libgap.Gap.global_context()` method and not construct objects of this class manually.

INPUT:

- variable – string. The variable name.
- value – anything that defines a GAP object.

EXAMPLES:
sage: libgap.set_global('FooBar', 1)

sage: with libgap.global_context('FooBar', 2):
....:   print(libgap.get_global('FooBar'))
2

sage: libgap.get_global('FooBar')
1

8.2 Common global functions defined by GAP.

8.3 Long tests for GAP

These stress test the garbage collection inside GAP

sage.libs.gap.test_long.test_loop_1()

EXAMPLES:

sage: from sage.libs.gap.test_long import test_loop_1
sage: test_loop_1()  # long time (up to 25s on sage.math, 2013)

sage.libs.gap.test_long.test_loop_2()

EXAMPLES:

sage: from sage.libs.gap.test_long import test_loop_2
sage: test_loop_2()  # long time (10s on sage.math, 2013)

sage.libs.gap.test_long.test_loop_3()

EXAMPLES:

sage: from sage.libs.gap.test_long import test_loop_3
sage: test_loop_3()  # long time (31s on sage.math, 2013)

8.4 Utility functions for GAP

exception sage.libs.gap.util.GAPError

Bases: exceptions.ValueError

Exceptions raised by the GAP library

class sage.libs.gap.util.ObjWrapper

Bases: object

Wrapper for GAP master pointers

EXAMPLES:

sage: from sage.libs.gap.util import ObjWrapper
sage: x = ObjWrapper()
sage: y = ObjWrapper()
sage: x == y
True

sage.libs.gap.util.gap_root()

Find the location of the GAP root install which is stored in the gap startup script.
EXAMPLES:

```python
sage: from sage.libs.gap.util import gap_root
sage: gap_root()  # random output
'/home/vbraun/opt/sage-5.3.rc0/local/gap/latest'
```

```
sage.libs.gap.util.get_owned_objects()  
Helper to access the refcount dictionary from Python code
```

## 8.5 Library Interface to GAP

This module implements a fast C library interface to GAP. To use it, you simply call `libgap` (the parent of all `GapElement` instances) and use it to convert Sage objects into GAP objects.

### EXAMPLES:

```python
sage: a = libgap(10)
sage: a
10
sage: type(a)  # random output
<type 'sage.libs.gap.element.GapElement_Integer'>
sage: a*a
100
sage: timeit('a*a')  # random output
625 loops, best of 3: 898 ns per loop
```

Compared to the expect interface this is >1000 times faster:

```python
sage: b = gap('10')
sage: timeit('b*b')  # random output; long time
125 loops, best of 3: 2.05 ms per loop
```

If you want to evaluate GAP commands, use the `Gap.eval()` method:

```python
sage: libgap.eval('List([1..10], i->i^2)')
[ 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 ]
```

not to be confused with the `libgap` call, which converts Sage objects to GAP objects, for example strings to strings:

```python
sage: libgap('List([1..10], i->i^2)')
"List([1..10], i->i^2)"
```

You can usually use the `sage()` method to convert the resulting GAP element back to its Sage equivalent:

```python
sage: a.sage()  
10
sage: type(_)
<type 'sage.rings.integer.Integer'>
sage: libgap.eval('5/3 + 7*E(3)').sage()  
7*zeta3 + 5/3
sage: generators = gens_of_group.sage()
```

(continues on next page)
We can also specify which group in Sage the permutations should consider themselves as elements of when converted to Sage:

```
sage: A4 = groups.permutation.Alternating(4)
sage: generators = gens_of_group.sage(parent=A4); generators
[(1,2,3), (2,3,4)]
sage: all(gen.parent() is A4 for gen in generators)
True
```

So far, the following GAP data types can be directly converted to the corresponding Sage datatype:

1. GAP booleans `true`/`false` to Sage booleans `True`/`False`. The third GAP boolean value `fail` raises a `ValueError`.

2. GAP integers to Sage integers.

3. GAP rational numbers to Sage rational numbers.

4. GAP cyclotomic numbers to Sage cyclotomic numbers.

5. GAP permutations to Sage permutations.

6. The GAP containers `List` and `rec` are converted to Sage containers `list` and `dict`. Furthermore, the `sage()` method is applied recursively to the entries.

Special support is available for the GAP container classes. GAP lists can be used as follows:

```
sage: lst = libgap([1,5,7]); lst
[ 1, 5, 7 ]
sage: type(lst)
<type 'sage.libs.gap.element.GapElement_List'>
sage: len(lst)
3
sage: lst[0]
1
sage: [ x^2 for x in lst ]
[1, 25, 49]
sage: type(_[0])
<type 'sage.libs.gap.element.GapElement_Integer'>
```

Note that you can access the elements of GAP `List` objects as you would expect from Python (with indexing starting at 0), but the elements are still of type `GapElement`. The other GAP container type are records, which are similar to Python dictionaries. You can construct them directly from Python dictionaries:

```
sage: libgap({'a':123, 'b':456})
rec( a := 123, b := 456 )
```

Or get them as results of computations:

```
sage: rec = libgap.eval('rec(a:=123, b:=456, Sym3:=SymmetricGroup(3))')
sage: rec['Sym3']
```

(continues on next page)
Sym( [ 1 .. 3 ] )
\begin{verbatim}
sage: rec
{'Sym3': Sym( [ 1 .. 3 ] ), 'a': 123, 'b': 456}
\end{verbatim}

The output is a Sage dictionary whose keys are Sage strings and whose values are instances of `GapElement()`. So, for example, `rec['a']` is not a Sage integer. To recursively convert the entries into Sage objects, you should use the `sage()` method:

\begin{verbatim}
sage: rec.sage()
{'Sym3': NotImplementedError('cannot construct equivalent Sage object',),
 'a': 123,
 'b': 456}
\end{verbatim}

Now `rec['a']` is a Sage integer. We have not implemented the conversion of the GAP symmetric group to the Sage symmetric group yet, so you end up with a `NotImplementedError` exception object. The exception is returned and not raised so that you can work with the partial result.

While we don’t directly support matrices yet, you can convert them to Gap List of Lists. These lists are then easily converted into Sage using the recursive expansion of the `sage()` method:

\begin{verbatim}
sage: M = libgap.eval('BlockMatrix([[1,1,[[1, 2],[ 3, 4]]], [1,2,[[9,10],[11,12]]],
                   [2,2,[[5, 6],[ 7, 8]]],2,2))
sage: M
<block matrix of dimensions (2*2)x(2*2)>
sage: M.List()
# returns a GAP List of Lists
\[
\begin{bmatrix}
1 & 2 & 9 & 10 \\
3 & 4 & 11 & 12 \\
0 & 0 & 5 & 6 \\
0 & 0 & 7 & 8 \\
\end{bmatrix}
\]
sage: M.List().sage()
# returns a Sage list of lists
\[
\begin{bmatrix}
1 & 2 & 9 & 10 \\
3 & 4 & 11 & 12 \\
0 & 0 & 5 & 6 \\
0 & 0 & 7 & 8 \\
\end{bmatrix}
\]
sage: matrix(ZZ, _)
\[
\begin{bmatrix}
1 & 2 & 9 & 10 \\
3 & 4 & 11 & 12 \\
0 & 0 & 5 & 6 \\
0 & 0 & 7 & 8 \\
\end{bmatrix}
\]
\end{verbatim}

### 8.5.1 Using the GAP C library from Cython

**Todo:** Expand the following text

We are using the GAP API provided by the GAP project since GAP 4.10.

**AUTHORS:**

- Volker Braun (2012-08-28, GAP/Singular workshop): update to gap-4.5.5, make it ready for public consumption.
- Dima Pasechnik (2018-09-18, GAP Days): started the port to native libgap API

```python
class sage.libs.gap.libgap.Gap
    Bases: sage.structure.parent.Parent
    The libgap interpreter object.
```
Note: This object must be instantiated exactly once by the libgap. Always use the provided libgap instance, and never instantiate Gap manually.

**EXAMPLES:**

```python
sage: libgap.eval('SymmetricGroup(4)')
Sym([1 .. 4])
```

**Element**

alias of `sage.libs.gap.element.GapElement`

**collect()**

Manually run the garbage collector

**EXAMPLES:**

```python
sage: a = libgap(123)
sage: del a
sage: libgap.collect()
```

**count_GAP_objects()**

Return the number of GAP objects that are being tracked by GAP.

**OUTPUT:**

An integer

**EXAMPLES:**

```python
sage: libgap.count_GAP_objects()  # random output
5
```

**eval(gap_command)**

Evaluate a gap command and wrap the result.

**INPUT:**

- gap_command – a string containing a valid gap command without the trailing semicolon.

**OUTPUT:**

A GapElement.

**EXAMPLES:**

```python
sage: libgap.eval('0')
0
sage: libgap.eval('"string"')
"string"
```

**function_factory(function_name)**

Return a GAP function wrapper

This is almost the same as calling `libgap.eval(function_name)`, but faster and makes it obvious in your code that you are wrapping a function.

**INPUT:**

- function_name – string. The name of a GAP function.
OUTPUT:
A function wrapper `GapElement_Function` for the GAP function. Calling it from Sage is equivalent to calling the wrapped function from GAP.

EXAMPLES:

```python
sage: libgap.function_factory('Print')
<Gap function "Print">
```

`get_global(variable)`
Get a GAP global variable

INPUT:
- `variable` – string. The variable name.

OUTPUT:
A `GapElement` wrapping the GAP output. A `ValueError` is raised if there is no such variable in GAP.

EXAMPLES:

```python
sage: libgap.set_global('FooBar', 1)
sage: libgap.get_global('FooBar')
1
sage: libgap.unset_global('FooBar')
sage: libgap.get_global('FooBar')
Traceback (most recent call last):
... GAPError: Error, VAL_GVAR: No value bound to FooBar
```

`global_context(variable, value)`
Temporarily change a global variable

INPUT:
- `variable` – string. The variable name.
- `value` – anything that defines a GAP object.

OUTPUT:
A context manager that sets/reverts the given global variable.

EXAMPLES:

```python
sage: libgap.set_global('FooBar', 1)
sage: with libgap.global_context('FooBar', 2):
....:     print(libgap.get_global('FooBar'))
2
sage: libgap.get_global('FooBar')
1
```

`mem()`
Return information about GAP memory usage

This method is deprecated and is a no-op. Use `Gap.show()` to display memory-usage and bag count statistics from GASMAN.

`one()`
Return (integer) one in GAP.

EXAMPLES:


```python
sage: libgap.one()
1
sage: parent(_)
C library interface to GAP
```

**set_global**(variable, value)

Set a GAP global variable

**INPUT:**

- variable – string. The variable name.
- value – anything that defines a GAP object.

**EXAMPLES:**

```python
sage: libgap.set_global('FooBar', 1)
sage: libgap.get_global('FooBar')
1
sage: libgap.unset_global('FooBar')
sage: libgap.get_global('FooBar')
Traceback (most recent call last):
...
GAPError: Error, VAL_GVAR: No value bound to FooBar
```

**set_seed**(seed=None)

Reseed the standard GAP pseudo-random sources with the given seed.

Uses a random seed given by `current_randstate().ZZ_seed()` if `seed=None`. Otherwise the seed should be an integer.

**EXAMPLES:**

```python
sage: libgap.set_seed(0)
0
sage: [libgap.Random(1, 10) for i in range(5)]
[2, 3, 3, 4, 2]
```

**show()**

Return statistics about the GAP owned object list

This includes the total memory allocated by GAP as returned by `libgap.eval('TotalMemoryAllocated()')`, as well as garbage collection / object count statistics as returned by `libgap.eval('GasmanStatistics')`, and finally the total number of GAP objects held by Sage as `GapElement` instances.

The value `livekb + deadkb` will roughly equal the total memory allocated for GAP objects (see `libgap.eval('TotalMemoryAllocated()')`).

**Note:** Slight complication is that we want to do it without accessing libgap objects, so we don’t create new GapElements as a side effect.

**EXAMPLES:**

```python
sage: a = libgap(123)
sage: b = libgap(456)
sage: c = libgap(789)
sage: del b
```
Sage Reference Manual: C/C++ Library Interfaces, Release 8.7

(continued from previous page)

```python
sage: libgap.collect()
sage: libgap.show()  # random output
{'gasman_stats': {'full': {'cumulative': 110,
    'deadbags': 321400,
    'deadkb': 12967,
    'freekb': 15492,
    'livebags': 396645,
    'livekb': 37730,
    'time': 110,
    'totalkb': 65536},
    'nfull': 1,
    'npartial': 1},
    'nelements': 23123,
    'total_alloc': 3234234}
```

**unset_global** *(variable)*
Remove a GAP global variable

**INPUT:**
- ```variable```– string. The variable name.

**EXAMPLES:**

```python
sage: libgap.set_global('FooBar', 1)
sage: libgap.get_global('FooBar')
1
sage: libgap.unset_global('FooBar')
sage: libgap.get_global('FooBar')
Traceback (most recent call last):
  ... GAPError: Error, VAL_GVAR: No value bound to FooBar
```

**zero()**
Return (integer) zero in GAP.

**OUTPUT:**
A GapElement.

**EXAMPLES:**

```python
sage: libgap.zero()
0
```

### 8.6 Short tests for GAP

**sage.libs.gap.test.test_write_to_file()**
Test that libgap can write to files

See trac ticket #16502, trac ticket #15833.

**EXAMPLES:**

```python
sage: from sage.libs.gap.test import test_write_to_file
sage: test_write_to_file()
```
8.7 GAP element wrapper

This document describes the individual wrappers for various GAP elements. For general information about GAP, you should read the `libgap` module documentation.

```python
class sage.libs.gap.element.GapElement
    Bases: sage.structure.element.RingElement
    
    Wrapper for all Gap objects.
    
    **Note:** In order to create `GapElement`s you should use the `libgap` instance (the parent of all Gap elements) to convert things into `GapElement`. You must not create `GapElement` instances manually.
    
    **EXAMPLES:**
    
    ```python
    sage: libgap(0)
    0
    ```
    
    If Gap finds an error while evaluating, a `GAPError` exception is raised:
    
    ```python
    sage: libgap.eval('1/0')
    Traceback (most recent call last):
    ...
    GAPError: Error, Rational operations: <divisor> must not be zero
    ```
    
    Also, a `GAPError` is raised if the input is not a simple expression:
    
    ```python
    sage: libgap.eval('1; 2; 3')
    Traceback (most recent call last):
    ...
    GAPError: can only evaluate a single statement
    ```
    
    **deepcopy** *(mut)*
    
    Return a deepcopy of this Gap object
    
    **Note** that this is the same thing as calling `StructuralCopy` but much faster.
    
    **INPUT:**
    
    - **mut** *(boolean)* whehter to return an mutable copy
    
    **EXAMPLES:**
    
    ```python
    sage: a = libgap([[0,1],[2,3]])
sage: b = a.deepcopy(1)
sage: b[0,0] = 5
    sage: a
    [[ 0, 1 ], [ 2, 3 ]]
sage: b
    [[ 5, 1 ], [ 2, 3 ]]
sage: l = libgap([0,1])
sage: l.deepcopy(0).IsMutable()  # false
    sage: l.deepcopy(1).IsMutable()  # true
    ```
is_bool()  
Return whether the wrapped GAP object is a GAP boolean.

OUTPUT:  
Boolean.

EXAMPLES:

```sage
sage: libgap(True).is_bool()
True
```

is_function()  
Return whether the wrapped GAP object is a function.

OUTPUT:  
Boolean.

EXAMPLES:

```sage
sage: a = libgap.eval("NormalSubgroups")
sage: a.is_function()
True
sage: a = libgap(2/3)
sage: a.is_function()
False
```

is_list()  
Return whether the wrapped GAP object is a GAP List.

OUTPUT:  
Boolean.

EXAMPLES:

```sage
sage: libgap.eval('[1, 2,,5]').is_list()
True
sage: libgap.eval('3/2').is_list()
False
```

is_permutation()  
Return whether the wrapped GAP object is a GAP permutation.

OUTPUT:  
Boolean.

EXAMPLES:

```sage
sage: perm = libgap.PermList( libgap([1,5,2,3,4]) ); perm
(2,5,4,3)
sage: perm.is_permutation()
True
sage: libgap('this is a string').is_permutation()
False
```

is_record()  
Return whether the wrapped GAP object is a GAP record.

OUTPUT:
Boolean.

EXAMPLES:

```
sage: libgap.eval('[1, 2,,, 5]').is_record()
False
sage: libgap.eval('rec(a:=1, b:=3)').is_record()
True
```

is_string()

Return whether the wrapped GAP object is a GAP string.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: libgap('this is a string').is_string()
True
```

sage()

Return the Sage equivalent of the GapElement

EXAMPLES:

```
sage: libgap(1).sage()
1
sage: type(_)
<type 'sage.rings.integer.Integer'>

sage: libgap(3/7).sage()
3/7
sage: type(_)
<type 'sage.rings.rational.Rational'>

sage: libgap.eval('5 + 7*E(3)').sage()
7*zeta3 + 5

sage: libgap(Infinity).sage()
+Infinity
sage: libgap(-Infinity).sage()
-Infinity

sage: libgap(True).sage()
True
sage: libgap(False).sage()
False
sage: type(_)
<... 'bool'>

sage: libgap('this is a string').sage()
'this is a string'
```

(continues on next page)
\[ x^2 - 2x + 3 \]

```
sage: p.sage().parent()
Univariate Polynomial Ring in x over Integer Ring
```

```
sage: p = x^-2 + 3*x
sage: p.sage()
x^-2 + 3*x
```

```
sage: p = (3 * x^2 + x) / (x^2 - 2)
sage: p.sage().parent()
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
```

#### class sage.libs.gap.element.GapElement_Boolean

**Bases:** `sage.libs.gap.element.GapElement`  

Derived class of GapElement for GAP boolean values.

**EXAMPLES:**

```
sage: b = libgap(True)
sage: type(b)
<type 'sage.libs.gap.element.GapElement_Boolean'>
```

**sage()**  
Return the Sage equivalent of the `GapElement`

**OUTPUT:**  
A Python boolean if the values is either true or false. GAP booleans can have the third value `Fail`, in which case a `ValueError` is raised.

**EXAMPLES:**

```
sage: b = libgap.eval('true'); b
true
sage: type(_)
<type 'sage.libs.gap.element.GapElement_Boolean'>
sage: b.sage()
True
sage: type(_)
<... 'bool'>
```

```
sage: libgap.eval('fail')
fail
sage: _.sage()  
Traceback (most recent call last):
...
ValueError: the GAP boolean value "fail" cannot be represented in Sage
```

#### class sage.libs.gap.element.GapElement_Cyclotomic

**Bases:** `sage.libs.gap.element.GapElement`  

Derived class of GapElement for GAP universal cyclotomics.

**EXAMPLES:**
sage: libgap.eval('E(3)')
E(3)
sage: type(_)
<type 'sage.libs.gap.element.GapElement_Cyclotomic'>

```
sage (ring=None)
    Return the Sage equivalent of the GapElement_Cyclotomic.

    INPUT:
    • ring – a Sage cyclotomic field or None (default). If not specified, a suitable minimal cyclotomic field will be constructed.

    OUTPUT:
    A Sage cyclotomic field element.

    EXAMPLES:

    sage: n = libgap.eval('E(3)')
sage: n.sage()  
zeta3
sage: parent(_)
Cyclotomic Field of order 3 and degree 2
sage: n.sage(ring=CyclotomicField(6))
zeta6 - 1
sage: libgap.E(3).sage(ring=CyclotomicField(3))
zeta3
sage: libgap.E(3).sage(ring=CyclotomicField(6))
zeta6 - 1
```

```
class sage.libs.gap.element.GapElement_FiniteField
    Bases: sage.libs.gap.element.GapElement

    Derived class of GapElement for GAP finite field elements.

    EXAMPLES:

    sage: libgap.eval('Z(5)^2')
Z(5)^2
sage: type(_)
<type 'sage.libs.gap.element.GapElement_FiniteField'>

lift()
    Return an integer lift.

    OUTPUT:
    The smallest positive GapElement_Integer that equals self in the prime finite field.

    EXAMPLES:

    sage: n = libgap.eval('Z(5)^2')
sage: n.lift()  
4
sage: type(_)
<type 'sage.libs.gap.element.GapElement_Integer'>
```
```
sage: n = libgap.eval('Z(25)')
sage: n.lift()
Traceback (most recent call last):
  TypeError: not in prime subfield
```

`sage (ring=None, var='a')`  
Return the Sage equivalent of the `GapElement_FiniteField`.

**INPUT:**

- `ring` – a Sage finite field or `None` (default). The field to return `self` in. If not specified, a suitable finite field will be constructed.

**OUTPUT:**

An Sage finite field element. The isomorphism is chosen such that the Gap `PrimitiveRoot()` maps to the Sage `multiplicative_generator()`.

**EXAMPLES:**

```
sage: n = libgap.eval('Z(25)^2')
sage: n.sage()
a + 3
sage: parent(_)
Finite Field in a of size 5^2
```

```
sage: n.sage(ring=GF(5))
Traceback (most recent call last):
  ...ValueError: the given ring is incompatible ...
```

```
class sage.libs.gap.element.GapElement_Float
    Bases: sage.libs.gap.element.GapElement

    Derived class of GapElement for GAP floating point numbers.

    **EXAMPLES:**

    ```
sage: i = libgap(123.5)
sage: type(i)
<type 'sage.libs.gap.element.GapElement_Float'>
sage: RDF(i)
123.5
```
```

```
sage (ring=None)
    Return the Sage equivalent of the `GapElement_Float`.

    **INPUT:**

    - `ring` – a floating point field or `None` (default). If not specified, the default Sage RDF is used.

    **OUTPUT:**

    A Sage double precision floating point number

    **EXAMPLES:**

    ```
sage: a = libgap.eval("Float(3.25)").sage()
sage: a
3.25
```
```
class sage.libs.gap.element.GapElement_Function
Bases: sage.libs.gap.element.GapElement

Derived class of GapElement for GAP functions.

EXAMPLES:

sage: f = libgap.Cycles
sage: type(f)
<type 'sage.libs.gap.element.GapElement_Function'>

class sage.libs.gap.element.GapElement_Integer
Bases: sage.libs.gap.element.GapElement

Derived class of GapElement for GAP integers.

EXAMPLES:

sage: i = libgap(123)
sage: type(i)
<type 'sage.libs.gap.element.GapElement_Integer'>
sage: ZZ(i)
123

is_C_int()

Return whether the wrapped GAP object is a immediate GAP integer.

An immediate integer is one that is stored as a C integer, and is subject to the usual size limits. Larger integers are stored in GAP as GMP integers.

OUTPUT:

Boolean.

EXAMPLES:

sage: n = libgap(1)
sage: type(n)
<type 'sage.libs.gap.element.GapElement_Integer'>
sage: n.is_C_int()
True
sage: n.IsInt()
true
sage: N = libgap(2^130)
sage: type(N)
<type 'sage.libs.gap.element.GapElement_Integer'>
sage: N.is_C_int()
False
sage: N.IsInt()
true

sage (ring=None)

Return the Sage equivalent of the GapElement_Integer

- ring – Integer ring or None (default). If not specified, a the default Sage integer ring is used.
A Sage integer

EXAMPLES:

```
sage: libgap([ 1, 3, 4 ]).sage()
[1, 3, 4]
sage: all( x in ZZ for x in _ )
True
sage: libgap(132).sage(ring=IntegerModRing(13))
2
sage: parent(_)
Ring of integers modulo 13
```

class sage.libs.gap.element.GapElement_IntegerMod
_bases: sage.libs.gap.element.GapElement

Derived class of GapElement for GAP integers modulo an integer.

EXAMPLES:

```
sage: n = IntegerModRing(123)(13)
sage: i = libgap(n)
sage: type(i)
<type 'sage.libs.gap.element.GapElement_IntegerMod'>
```

**lift()**

Return an integer lift.

OUTPUT:

A GapElement_Integer that equals self in the integer mod ring.

EXAMPLES:

```
sage: n = libgap.eval('One(ZmodnZ(123)) * 13')
sage: n.lift()
13
```

**sage** *(ring=None)*

Return the Sage equivalent of the GapElement_IntegerMod

INPUT:

- ring – Sage integer mod ring or None (default). If not specified, a suitable integer mod ring is used automatically.

OUTPUT:

A Sage integer modulo another integer.

EXAMPLES:

```
sage: n = libgap.eval('One(ZmodnZ(123)) * 13')
sage: n.sage()
13
sage: parent(_)
Ring of integers modulo 123
```
**class** sage.libs.gap.element.GapElement_List

**Bases:** sage.libs.gap.element.GapElement

Derived class of GapElement for GAP Lists.

**Note:** Lists are indexed by 0..len(l) − 1, as expected from Python. This differs from the GAP convention where lists start at 1.

**EXAMPLES:**

```python
sage: lst = libgap.SymmetricGroup(3).List(); lst
[ (), (1,3), (1,2,3), (2,3), (1,3,2), (1,2) ]
sage: type(lst)
<type 'sage.libs.gap.element.GapElement_List'>
sage: len(lst)
6
sage: lst[3]
(2,3)
```

We can easily convert a Gap List object into a Python list:

```python
sage: list(lst)
[(), (1,3), (1,2,3), (2,3), (1,3,2), (1,2)]
sage: type(_)
<... 'list'>
```

Range checking is performed:

```python
sage: lst[10]
Traceback (most recent call last):
  ...
IndexError: index out of range.
```

**matrix**(ring=None)

Return the list as a matrix.

GAP does not have a special matrix data type, they are just lists of lists. This function converts a GAP list of lists to a Sage matrix.

**OUTPUT:**

A Sage matrix.

**EXAMPLES:**

```python
sage: F = libgap.GF(4)
sage: a = F.PrimitiveElement()
sage: m = libgap([[[a,a^0],[0*a,a^2]]]); m
[ [ Z(2^2), Z(2)^0 ],
  [ 0*Z(2), Z(2^2)^2 ] ]
sage: m.IsMatrix()
true
sage: matrix(m)
[  a  1 ]
[ 0 a + 1 ]
sage: matrix(GF(4,'B'), m)
[  B  1 ]
[ 0 B + 1]
```

(continues on next page)
sage: M = libgap.eval('SL(2,GF(5))').GeneratorsOfGroup()[1]
sage: type(M)
<type 'sage.libs.gap.element.GapElement_List'>
sage: M[0][0]
Z(5)^2
sage: M.IsMatrix()
true
sage: M.matrix()
\[[4 1]
 [4 0]\]

\textbf{\texttt{sage}(**\texttt{kwds})}
Return the Sage equivalent of the \texttt{GapElement}

\textbf{OUTPUT:}
A Python list.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: libgap([ 1, 3, 4 ]).sage()
[1, 3, 4]
sage: all( x in ZZ for x in _ )
True
\end{verbatim}

\textbf{\texttt{vector}(ring=None)}
Return the list as a vector.

GAP does not have a special vector data type, they are just lists. This function converts a GAP list to a Sage vector.

\textbf{OUTPUT:}
A Sage vector.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: F = libgap.GF(4)
sage: a = F.PrimitiveElement()
sage: m = libgap([0*a, a, a^3, a^2]); m
\[[ 0*Z(2), Z(2^2), Z(2)^0, Z(2^2)^2 ]
sage: type(m)
<type 'sage.libs.gap.element.GapElement_List'>
sage: m[3]
Z(2^2)^2
sage: vector(m)
(0, a, 1, a + 1)
sage: vector(GF(4,'B'), m)
(0, B, 1, B + 1)
\end{verbatim}

\textbf{\texttt{class} }\texttt{sage.libs.gap.element.GapElement\_MethodProxy}
\textbf{Bases:} \texttt{sage.libs.gap.element.GapElement\_Function}

\textbf{Helper class returned by} \texttt{GapElement.__getattr__}.

\textbf{Derived class of} \texttt{GapElement} for GAP functions. Like its parent, you can call instances to implement function call syntax. The only difference is that a fixed first argument is prepended to the argument list.

\textbf{EXAMPLES:}
```python
sage: lst = libgap([])
sage: lst.Add
<Gap function "Add">  
sage: type(_)
<type 'sage.libs.gap.element.GapElement_MethodProxy'>
sage: lst.Add(1)
sage: lst
[ 1 ]
```

**class** `sage.libs.gap.element.GapElement_Permutation`  
**Bases:** `sage.libs.gap.element.GapElement`  

Derived class of `GapElement` for GAP permutations.

**Note:** Permutations in GAP act on the numbers starting with 1.

**EXAMPLES:**

```python
sage: perm = libgap.eval('(1,5,2)(4,3,8)')
sage: type(perm)
<type 'sage.libs.gap.element.GapElement_Permutation'>
sage: (parent=None)
Return the Sage equivalent of the `GapElement`.  
If the permutation group is given as parent, this method is *much* faster.

**EXAMPLES:**

```python
sage: perm_gap = libgap.eval('(1,5,2)(4,3,8)'); perm_gap
(1,5,2)(3,8,4)
sage: perm_gap.sage()
[5, 1, 8, 3, 2, 6, 7, 4]
sage: type(_)
<class 'sage.combinat.permutation.StandardPermutations_all_with_category.˓
   element_class'>
sage: perm_gap.sage(PermutationGroup([(1,2),(1,2,3,4,5,6,7,8)]))
(1,5,2)(3,8,4)
sage: type(_)
<type 'sage.groups.perm_gps.permgroup_element.PermutationGroupElement'>
```

**class** `sage.libs.gap.element.GapElement_Rational`  
**Bases:** `sage.libs.gap.element.GapElement`  

Derived class of `GapElement` for GAP rational numbers.

**EXAMPLES:**

```python
sage: r = libgap(123/456)
sage: type(r)
<type 'sage.libs.gap.element.GapElement_Rational'>
sage: (ring=None)
Return the Sage equivalent of the `GapElement`.  

**INPUT:**

- `ring` – the Sage rational ring or `None` (default). If not specified, the rational ring is used automatically.

94 Chapter 8. GAP
OUTPUT:

A Sage rational number.

EXAMPLES:

```python
sage: r = libgap(123/456); r
41/152
sage: type(_)
<type 'sage.libs.gap.element.GapElement_Rational'>
sage: r.sage()
41/152
sage: type(_)
<type 'sage.rings.rational.Rational'>
```

```python
class sage.libs.gap.element.GapElement_Record

Bases: sage.libs.gap.element.GapElement

Derived class of GapElement for GAP records.

EXAMPLES:

```python
sage: rec = libgap.eval('rec(a:=123, b:=456)')
sage: type(rec)
<type 'sage.libs.gap.element.GapElement_Record'>
sage: len(rec)
2
sage: rec['a']
123
```

We can easily convert a Gap `rec` object into a Python `dict`:

```python
sage: dict(rec)
{'a': 123, 'b': 456}
sage: type(_)
<... 'dict'>
```

Range checking is performed:

```python
sage: rec['no_such_element']
Traceback (most recent call last):
...
GAPError: Error, Record Element: '<rec>.no_such_element' must have an assigned value
```

```python
record_name_to_index(name)

Convert string to GAP record index.

INPUT:

- `py_name` – a python string.

OUTPUT:

A UInt, which is a GAP hash of the string. If this is the first time the string is encountered, a new integer is returned(!)

EXAMPLES:
sage: rec = libgap.eval('rec(first:=123, second:=456)')
sage: rec.record_name_to_index('first')  # random output
1812L
sage: rec.record_name_to_index('no_such_name')  # random output
3776L

sage()

Return the Sage equivalent of the GapElement.

EXAMPLES:

sage: libgap.eval('rec(a:=1, b:=2)').sage()
{'a': 1, 'b': 2}
sage: all( isinstance(key,str) and val in ZZ for key,val in _.items() )
True
sage: rec = libgap.eval('rec(a:=123, b:=456, Sym3:=SymmetricGroup(3))')
sage: rec.sage()
{'Sym3': NotImplementedError('cannot construct equivalent Sage object'),
'a': 123,
'b': 456}

class sage.libs.gap.element.GapElement_RecordIterator

Bases: object

Iterator for GapElement_Record

Since Cython does not support generators yet, we implement the older iterator specification with this auxiliary class.

INPUT:

* rec  the GapElement_Record to iterate over.

EXAMPLES:

sage: rec = libgap.eval('rec(a:=123, b:=456)')
sage: sorted(rec)
[('a', 123), ('b', 456)]
sage: dict(rec)
{'a': 123, 'b': 456}

next()

x.next() -> the next value, or raise StopIteration

class sage.libs.gap.element.GapElement_Ring

Bases: sage.libs.gap.element.GapElement

Derived class of GapElement for GAP rings (parents of ring elements).

EXAMPLES:

sage: i = libgap(ZZ)
sage: type(i)
<type 'sage.libs.gap.element.GapElement_Ring'>

ring_cyclotomic()

Construct an integer ring.

EXAMPLES:
sage: libgap.CyclotomicField(6).ring_cyclotomic()
Cyclotomic Field of order 3 and degree 2

**ring Finite Field**(var='a')

Construct an integer ring.

**EXAMPLES:**

sage: libgap.GF(3,2).ring_finite_field(var='A')
Finite Field in A of size 3^2

**ring Integer**()

Construct the Sage integers.

**EXAMPLES:**

sage: libgap.eval('Integers').ring_integer()
Integer Ring

**ring Integer Mod**()

Construct a Sage integer mod ring.

**EXAMPLES:**

sage: libgap.eval('ZmodnZ(15)').ring_integer_mod()
Ring of integers modulo 15

**ring Polynomial**()

Construct a polynomial ring.

**EXAMPLES:**

sage: B = libgap(QQ['x'])
sage: B.ring_polynomial()
Univariate Polynomial Ring in x over Rational Field

sage: B = libgap(ZZ['x','y'])
sage: B.ring_polynomial()
Multivariate Polynomial Ring in x, y over Integer Ring

**ring Rational**()

Construct the Sage rationals.

**EXAMPLES:**

sage: libgap.eval('Rationals').ring_rational()
Rational Field

**sage(***kwds)**

Return the Sage equivalent of the `GapElement_Ring`.

**INPUT:**

- **kwds** – keywords that are passed on to the `ring_` method.

**OUTPUT:**

A Sage ring.

**EXAMPLES:**
sage: libgap.eval('Integers').sage()
Integer Ring

sage: libgap.eval('Rationals').sage()
Rational Field

sage: libgap.eval('ZmodnZ(15)').sage()
Ring of integers modulo 15

sage: libgap.GF(3,2).sage(var='A')
Finite Field in A of size 3^2

sage: libgap.CyclotomicField(6).sage()
Cyclotomic Field of order 3 and degree 2

sage: libgap(QQ['x','y']).sage()
Multivariate Polynomial Ring in x, y over Rational Field

class sage.libs.gap.element.GapElement_String
Bases: sage.libs.gap.element.GapElement

Derived class of GapElement for GAP strings.

EXAMPLES:

sage: s = libgap('string')
sage: type(s)
<type 'sage.libs.gap.element.GapElement_String'>
sage: s
"string"
sage: print(s)
string

sage()
Convert this GapElement_String to a Python string.

OUTPUT:
A Python string.

EXAMPLES:

sage: s = libgap.eval(' "string" '); s
"string"
sage: type(_)
<type 'sage.libs.gap.element.GapElement_String'>
sage: str(s)
'string'
sage: s.sage()
'string'
sage: type(_)
<type 'str'>

8.8 LibGAP Workspace Support

The single purpose of this module is to provide the location of the libgap saved workspace and a time stamp to invalidate saved workspaces.
sage.libs.gap.saved_workspace.timestamp()

Return a time stamp for (lib)gap

OUTPUT:

Float. Unix timestamp of the most recently changed GAP/LibGAP file(s). In particular, the timestamp increases whenever a gap package is added.

EXAMPLES:

```
sage: from sage.libs.gap.saved_workspace import timestamp
sage: timestamp()
# random output
1406642467.25684
sage: type(timestamp())
<... 'float'>
```

sage.libs.gap.saved_workspace.workspace(name='workspace')

Return the filename of the gap workspace and whether it is up to date.

INPUT:

* name – string. A name that will become part of the workspace filename.

OUTPUT:

Pair consisting of a string and a boolean. The string is the filename of the saved libgap workspace (or that it should have if it doesn’t exist). The boolean is whether the workspace is up-to-date. You may use the workspace file only if the boolean is True.

EXAMPLES:

```
sage: from sage.libs.gap.saved_workspace import workspace
sage: ws, up_to_date = workspace()
sage: ws
'/.../gap/libgap-workspace-...'
sage: isinstance(up_to_date, bool)
True
```
9.1 Interface between flint matrices and linbox

This module only contains C++ code (and the interface is fully C compatible). It basically contains what used to be in the LinBox source code under interfaces/sage/linbox-sage.C written by M. Albrecht and C. Pernet. The functions available are:

- void linbox_fmpz_mat_mul(fmpz_mat_t C, fmpz_mat_t A, fmpz_mat_t B): set C to be the result of the multiplication A * B
- void linbox_fmpz_mat_charpoly(fmpz_poly_t cp, fmpz_mat_t A): set cp to be the characteristic polynomial of the square matrix A
- void linbox_fmpz_mat_minpoly(fmpz_poly_t mp, fmpz_mat_t A): set mp to be the minimal polynomial of the square matrix A
- unsigned long linbox_fmpz_mat_rank(fmpz_mat_t A): return the rank of the matrix A
- void linbox_fmpz_mat_det(fmpz_t det, fmpz_mat_t A): set det to the determinant of the square matrix A
10.1 An interface to Anders Buch’s Littlewood-Richardson Calculator

lrcalc

The “Littlewood-Richardson Calculator” is a C library for fast computation of Littlewood-Richardson (LR) coefficients and products of Schubert polynomials. It handles single LR coefficients, products of and coproducts of Schur functions, skew Schur functions, and fusion products. All of the above are achieved by counting LR (skew)-tableaux (also called Yamanouchi (skew)-tableaux) of appropriate shape and content by iterating through them. Additionally, lrcalc handles products of Schubert polynomials.

The web page of lrcalc is http://sites.math.rutgers.edu/~asbuch/lrcalc/.

The following describes the Sage interface to this library.

EXAMPLES:

```sage
sage: import sage.libs.lrcalc.lrcalc as lrcalc

Compute a single Littlewood-Richardson coefficient:

sage: lrcalc.lrcoef([3,2,1],[2,1],[2,1])
2

Compute a product of Schur functions; return the coefficients in the Schur expansion:

sage: lrcalc.mult([2,1], [2,1])
{[2, 2, 2]: 1, [3, 2, 1]: 2, [3, 3]: 1, [4, 1, 1]: 1, [4, 2]: 1}

Same product, but include only partitions with at most 3 rows. This corresponds to computing in the representation ring of gl(3):

sage: lrcalc.mult([2,1], [2,1], 3)
{[2, 2, 2]: 1, [3, 2, 1]: 2, [3, 3]: 1, [4, 1, 1]: 1, [4, 2]: 1}

We can also compute the fusion product, here for sl(3) and level 2:

sage: lrcalc.mult([3,2,1], [3,2,1], 3, 2)
{[4, 4, 4]: 1, [5, 4, 3]: 1}
```
Compute the expansion of a skew Schur function:

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

Compute the coproduct of a Schur function:

```sage
lrcalc.coprod([3,2,1])
{([1, 1, 1], [2, 1]): 1, ([2, 1], [2, 1]): 2, ([2, 1], [3]): 1, ([2, 1, 1], [1, 1]): 1, ([2, 1, 1], [2]): 1, ([2, 2], [1, 1]): 1, ([2, 2], [2]): 1, ([2, 2], [1]): 1, ([3, 1], [1, 1]): 1, ([3, 1], [2]): 1, ([3, 1, 1], [1]): 1, ([3, 2], [1]): 1, ([3, 2, 1], []): 1}
```

Multiply two Schubert polynomials:

```sage
lrcalc.mult_schubert([4,2,1,3], [1,4,2,5,3])
{[4, 5, 1, 3, 2]: 1, [5, 3, 1, 4, 2]: 1, [5, 4, 1, 2, 3]: 1, [6, 2, 1, 4, 3, 5]: 1}
```

Same product, but include only permutations of 5 elements in the result. This corresponds to computing in the cohomology ring of Fl(5):

```sage
lrcalc.mult_schubert([4,2,1,3], [1,4,2,5,3], 5)
{[4, 5, 1, 3, 2]: 1, [5, 3, 1, 4, 2]: 1, [5, 4, 1, 2, 3]: 1}
```

List all Littlewood-Richardson tableaux of skew shape $\mu/\nu$; in this example $\mu = [3, 2, 1]$ and $\nu = [2, 1]$. Specifying a third entry `maxrows` restricts the alphabet to $\{1, 2, \ldots, \text{maxrows}\}$:

```sage
list(lrcalc.lrskew([3,2,1],[2,1]))
[[[None, None, 1], [None, 1], [1]], [[None, None, 1], [None, 1], [2]],
[[None, None, 1], [None, 2], [1]], [[None, None, 1], [None, 2], [3]]]
```

```sage
list(lrcalc.lrskew([3,2,1],[2,1],maxrows=2))
[[[None, None, 1], [None, 1], [1]], [[None, None, 1], [None, 1], [2]], [[None, None, ...
-1], [None, 2], [1]]]
```

Todo: use this library in the `SymmetricFunctions` code, to make it easy to apply it to linear combinations of Schur functions.

See also:

- `lrcoef()`
- `mult()`
- `coprod()`
• *skew()
• *lrskew()
• *mult_schubert()

**Underlying algorithmic in lrcalc**

Here is some additional information regarding the main low-level C-functions in *lrcalc*. Given two partitions *outer* and *inner* with *inner* contained in *outer*, the function:

```c
skewtab *st_new(vector *outer, vector *inner, vector *conts, int maxrows)
```

constructs and returns the (lexicographically) first LR skew tableau of shape *outer / inner*. Further restrictions can be imposed using *conts* and *maxrows*.

Namely, the integer *maxrows* is a bound on the integers that can be put in the tableau. The name is chosen because this will limit the partitions in the output of *skew()* or *mult()* to partitions with at most this number of rows.

The vector *conts* is the content of an empty tableau(!!). More precisely, this vector is added to the usual content of a tableau whenever the content is needed. This affects which tableaux are considered LR tableaux (see *mult()* below). *conts* may also be the NULL pointer, in which case nothing is added.

The other function:

```c
int *st_next(skewtab *st)
```

computes in place the (lexicographically) next skew tableau with the same constraints, or returns 0 if *st* is the last one.

For a first example, see the *skew()* function code in the *lrcalc* source code. We want to compute a skew Schur function, so create a skew LR tableau of the appropriate shape with *st_new* (with *conts* = NULL), then iterate through all the LR tableaux with *st_next*(). For each skew tableau, we use that *st-*->*conts* is the content of the skew tableau, find this shape in the res hash table and add one to the value.

For a second example, see *mult(vector *sh1, vector *sh2, maxrows)*. Here we call *st_new()* with the shape *sh1 / (0)* and use *sh2* as the *conts* argument. The effect of using *sh2* in this way is that *st_next* will iterate through semistandard tableaux *T* of shape *sh1* such that the following tableau:

```
111111
222222 <--- minimal tableau of shape sh2
333
*****
**T**
****
**
```

is a LR skew tableau, and *st-*->*conts* contains the content of the combined tableaux.

More generally, *st_new(outer, inner, conts, maxrows)* and *st_next* can be used to compute the Schur expansion of the product $S_\{(outer/inner) \ast S_conts\}$, restricted to partitions with at most *maxrows* rows.

**AUTHORS:**

- Mike Hansen (2010): core of the interface
- Anne Schilling, Nicolas M. Thiéry, and Anders Buch (2011): fusion product, iterating through LR tableaux, finalization, documentation
sage.libs.lrcalc.lrcalc.coprod\(\text{part, all=0}\)

Compute the coproduct of a Schur function.

Return a linear combination of pairs of partitions representing the coproduct of the Schur function given by the partition part.

INPUT:

- part – a partition.
- all – an integer.

If all is non-zero then all terms are included in the result. If all is zero, then only pairs of partitions (part1, part2) for which the weight of part1 is greater than or equal to the weight of part2 are included; the rest of the coefficients are redundant because Littlewood-Richardson coefficients are symmetric.

EXAMPLES:

```python
sage: from sage.libs.lrcalc.lrcalc import coprod
sage: sorted(coprod([2,1]).items())
[(([1, 1], [1]), 1), ([2], [1]), ([2, 1], []), ([2, 1], [1])]
```

sage.libs.lrcalc.lrcalc.lrcoef\(outer, inner1, inner2\)

Compute a single Littlewood-Richardson coefficient.

Return the coefficient of outer in the product of the Schur functions indexed by inner1 and inner2.

INPUT:

- outer – a partition (weakly decreasing list of non-negative integers).
- inner1 – a partition.
- inner2 – a partition.

Note: This function converts its inputs into Partition()’s. If you don’t need these checks and your inputs are valid, then you can use lrcoef_unsafe().

EXAMPLES:

```python
sage: from sage.libs.lrcalc.lrcalc import lrcoef
sage: lrcoef([3,2,1], [2,1], [2,1])
2
sage: lrcoef([3,3], [2,1], [2,1])
1
sage: lrcoef([2,1,1,1,1], [2,1], [2,1])
0
```

sage.libs.lrcalc.lrcalc.lrcoef_unsafe\(outer, inner1, inner2\)

Compute a single Littlewood-Richardson coefficient.

Return the coefficient of outer in the product of the Schur functions indexed by inner1 and inner2.

INPUT:

- outer – a partition (weakly decreasing list of non-negative integers).
- inner1 – a partition.
- inner2 – a partition.
**Warning:** This function does not do any check on its input. If you want to use a safer version, use `lrcoef()`.

**EXAMPLES:**

```python
sage: from sage.libs.lrcalc.lrcalc import lrcoef_unsafe
sage: lrcoef_unsafe([3,2,1], [2,1], [2,1])
2
sage: lrcoef_unsafe([3,3], [2,1], [2,1])
1
sage: lrcoef_unsafe([2,1,1,1,1], [2,1], [2,1])
0
```

```python
sage.libs.lrcalc.lrcalc.lrskew(outer, inner, weight=None, maxrows=0)
```

Iterate over the skew LR tableaux of shape `outer / inner`.

**INPUT:**

- `outer` – a partition
- `inner` – a partition
- `weight` – a partition (optional)
- `maxrows` – an integer (optional)

**OUTPUT:** an iterator of `SkewTableau`

Specifying `maxrows` restricts the alphabet to `{1,2,...,maxrows}`.

Specifying `weight` returns only those tableaux of given content/weight.

**EXAMPLES:**

```python
sage: from sage.libs.lrcalc.lrcalc import lrskew
sage: for st in lrskew([3,2,1],[2]):
...: st.pp()
....:. . 1
1 1
2
....:. . 1
1 2
2
....:. . 1
1 2
3
sage: for st in lrskew([3,2,1],[2], maxrows=2):
...: st.pp()
....:. . 1
1 1
2
....:. . 1
1 2
2
sage: list(lrskew([3,2,1],[2], weight=[3,1]))
[[[None, None, 1], [1, 1], [2]]]
```
sage.libs.lrcalc.lrcalc.mult\(\text{part1, part2, maxrows=\text{None, level=\text{None, quantum=\text{None}}}}\)

Compute a product of two Schur functions.

Return the product of the Schur functions indexed by the partitions \text{part1} and \text{part2}.

\textbf{INPUT:}

\begin{itemize}
  \item \text{part1} – a partition
  \item \text{part2} – a partition
  \item \text{maxrows} – (optional) an integer
  \item \text{level} – (optional) an integer
  \item \text{quantum} – (optional) an element of a ring
\end{itemize}

If \text{maxrows} is specified, then only partitions with at most this number of rows are included in the result.

If both \text{maxrows} and \text{level} are specified, then the function calculates the fusion product for \text{sl(maxrows)} of the given level.

If \text{quantum} is set, then this returns the product in the quantum cohomology ring of the Grassmannian. In particular, both \text{maxrows} and \text{level} need to be specified.

\textbf{EXAMPLES:}

```
sage: from sage.libs.lrcalc.lrcalc import mult
sage: mult([2],[])
{(2): 1}

sage: sorted(mult([2],[2]).items())
{(2, 2): 1, (3, 1): 1, (4, 1): 1}

sage: sorted(mult([2],[2]).items())
{(2, 2, 1, 1): 1, (2, 2, 2): 1, (3, 1, 1, 1): 1, (3, 2, 1, 2): 1, (3, 3, 1, 2): 1, (4, 1, 1, 1): 1, (4, 2, 1): 1}

sage: sorted(mult([2],[2],maxrows=2).items())
{(3, 3): 1, (4, 2, 1): 1}

sage: mult([2],[2],rank=0)

sage: mult([2],[2],maxrows=2,level=2)
Traceback (most recent call last):
...
ValueError: maxrows needs to be specified if you specify the level

The quantum product::

sage: q = polygen(QQ, 'q')
sage: sorted(mult([1],[2], quantum=q).items())
{(1, q): 1, (2, 1): 1}

sage: sorted(mult([2],[2], quantum=q).items())
{(2, 2): 1, (3, 1, 1): 1, (4, 1, 1): 1, (5, 1, 1): 1}

sage: mult([2],[2], quantum=q)
Traceback (most recent call last):
...
ValueError: missing parameters maxrows or level
```

sage.libs.lrcalc.lrcalc.mult_schubert\(\text{w1, w2, rank=0}\)

Compute a product of two Schubert polynomials.
Return a linear combination of permutations representing the product of the Schubert polynomials indexed by the permutations \( w_1 \) and \( w_2 \).

**INPUT:**

- \( w_1 \) – a permutation.
- \( w_2 \) – a permutation.
- \( \text{rank} \) – an integer.

If \( \text{rank} \) is non-zero, then only permutations from the symmetric group \( S(\text{rank}) \) are included in the result.

**EXAMPLES:**

```python
sage: from sage.libs.lrcalc.lrcalc import mult_schubert
sage: result = mult_schubert([3, 1, 5, 2, 4], [3, 5, 2, 1, 4])
```

```python
sage: sorted(result.items())
```

```python
([[5, 4, 6, 1, 2, 3], 1), ([5, 6, 3, 1, 2, 4], 1), ([5, 7, 2, 1, 3, 4, 6], 1), ([6, 3, 5, 1, 2, 4], 1), ([6, 4, 3, 1, 2, 5], 1), ([6, 5, 2, 1, 3, 4], 1), ([7, 3, 4, 1, 2, 5, 6], 1), ([7, 4, 2, 1, 3, 5, 6], 1)]
```

### sage.libs.lrcalc.lrcalc.skew(outer, inner, maxrows=0)

**Compute the Schur expansion of a skew Schur function.**

Return a linear combination of partitions representing the Schur function of the skew Young diagram \( \text{outer} / \text{inner} \), consisting of boxes in the partition \( \text{outer} \) that are not in \( \text{inner} \).

**INPUT:**

- \( \text{outer} \) – a partition.
- \( \text{inner} \) – a partition.
- \( \text{maxrows} \) – an integer or None.

If \( \text{maxrows} \) is specified, then only partitions with at most this number of rows are included in the result.

**EXAMPLES:**

```python
sage: from sage.libs.lrcalc.lrcalc import skew
sage: sorted(skew([2,1],[1]).items())
```

```python
[((1, 1), 1), ([2, 1], 1)]
```

### sage.libs.lrcalc.lrcalc.test_iterable_to_vector(it)

**A wrapper function for the cdef function iterable_to_vector and vector_to_list, to test that they are working correctly.**

**EXAMPLES:**

```python
sage: from sage.libs.lrcalc.lrcalc import test_iterable_to_vector
sage: x = test_iterable_to_vector([3,2,1]); x
```

```python
[3, 2, 1]
```

### sage.libs.lrcalc.lrcalc.test_skewtab_to_SkewTableau(outer, inner)

**A wrapper function for the cdef function skewtab_to_SkewTableau for testing purposes.**

It constructs the first LR skew tableau of shape \( \text{outer}/\text{inner} \) as an lrcalcskewtab, and converts it to a SkewTableau.

**EXAMPLES:**
sage: from sage.libs.lrcalc.lrcalc import test_skewtab_to_SkewTableau
sage: test_skewtab_to_SkewTableau([3,2,1],[])
[[1, 1, 1], [2, 2], [3]]
sage: test_skewtab_to_SkewTableau([4,3,2,1],[1,1]).pp()
  1  1  1
  2  2
1  3
2
11.1 Utilities for Sage-mpmath interaction

Also patches some mpmath functions for speed

```
sage.libs.mpmath.utils.bitcount(n)
   Bitcount of a Sage Integer or Python int/long.
```

EXAMPLES:

```
sage: import mpmath.libmp import bitcount
call

sage: bitcount(0)
0
sage: bitcount(1)
1
sage: bitcount(100)
7
sage: bitcount(-100)
7
sage: bitcount(2r)
2
sage: bitcount(2L)
2
```

```
sage.libs.mpmath.utils.call(func, *args, **kwargs)
   Call an mpmath function with Sage objects as inputs and convert the result back to a Sage real or complex
   number.

   By default, a RealNumber or ComplexNumber with the current working precision of mpmath (mpmath.mp.prec)
   will be returned.

   If prec=n is passed among the keyword arguments, the temporary working precision will be set to n and the
   result will also have this precision.

   If parent=P is passed, P.prec() will be used as working precision and the result will be coerced to P (or the
   corresponding complex field if necessary).

   Arguments should be Sage objects that can be coerced into RealField or ComplexField elements. Arguments
   may also be tuples, lists or dicts (which are converted recursively), or any type that mpmath understands natively
   (e.g. Python floats, strings for options).

   EXAMPLES:

   ```
sage: import sage.libs.mpmath.all as a
sage: a.mp.prec = 53
```
sage: a.call(a.erf, 3+4*I)  
-120.186991395079 - 27.7503372936239*I  
sage: a.call(a.polylog, 2, 1/3+4/5*I)  
0.153548951541433 + 0.87511414299637*I  
sage: a.call(a.barnesg, 3+4*I)  
-0.000676375932234244 - 0.0000442236140124728*I  
sage: a.call(a.barnesg, -4)  
0.000000000000000  
sage: a.call(a.hyper, [2,3], [4,5], 1/3)  
1.10703578162508  
sage: a.call(a.hyper, [2,3], [4,(2,3)], 1/3)  
1.9576293509305  
sage: a.call(a.quad, a.erf, [0,1])  
0.486064958112256  
sage: a.call(a.gammainc, 3+4*I, 2/3, 1-pi*I, prec=100)  
-274.18871130777160922275236*I  
sage: x = (3+4*I).n(100)  
sage: y = (2/3).n(100)  
sage: z = (1-pi*I).n(100)  
sage: a.call(a.gammainc, x, y, z, prec=100)  
-274.18871130777160922275236*I  
sage: a.call(a.erf, infinity)  
1.00000000000000  
sage: a.call(a.erf, -infinity)  
-1.00000000000000  
sage: a.call(a.gamma, infinity)  
+infinity  
sage: a.call(a.polylog, 2, 1/2, parent=RR)  
0.582240526465012  
sage: a.call(a.polylog, 2, 2, parent=RR)  
2.467401100272344 - 2.1775860930360*I  
sage: a.call(a.polylog, 2, 1/2, parent=RealField(100))  
0.582240526465012  
sage: a.call(a.polylog, 2, 2, parent=RealField(100))  
2.467401100272339547086227500 - 2.17758609303602130506888982*I  
sage: a.call(a.polylog, 2, 1/2, parent=CC)  
0.582240526465012  
sage: type(_)
<type 'sage.rings.complex_number.ComplexNumber'>  
sage: a.call(a.polylog, 2, 1/2, parent=RDF)  
0.582240526465012  
sage: type(_)
<type 'sage.rings.real_double.RealDoubleElement'>

Check that trac ticket #11885 is fixed:

sage: a.call(a.ei, 1.0r, parent=float)  
1.8951178163559366

Check that trac ticket #14984 is fixed:

sage: a.call(a.log, -1.0r, parent=float)  
3.141592653589793j

sage.libs.mpmath.utils.from_man_exp(man, exp, prec=0, rnd='d')  
Create normalized mpf value tuple from mantissa and exponent.

With prec > 0, rounds the result in the desired direction if necessary.
EXAMPLES:

```python
sage: from mpmath.libmp import from_man_exp
sage: from_man_exp(-6, -1)
(1, 3, 0, 2)
sage: from_man_exp(-6, -1, 1, 'd')
(1, 1, 1, 1)
sage: from_man_exp(-6, -1, 1, 'u')
(1, 1, 2, 1)
```

```python
sage.libs.mpmath.utils.isqrt(n)
```

Square root (rounded to floor) of a Sage Integer or Python int/long. The result is a Sage Integer.

EXAMPLES:

```python
sage: from mpmath.libmp import isqrt
sage: isqrt(0)
0
sage: isqrt(100)
10
sage: isqrt(10)
3
sage: isqrt(10r)
3
sage: isqrt(10L)
3
```

```python
sage.libs.mpmath.utils.mpmath_to_sage(x, prec)
```

Convert any mpmath number (mpf or mpc) to a Sage RealNumber or ComplexNumber of the given precision.

EXAMPLES:

```python
sage: import sage.libs.mpmath.all as a
sage: a.mpmath_to_sage(a.mpf('2.5'), 53)
2.500000000000000
sage: a.mpmath_to_sage(a.mpc('2.5','-3.5'), 53)
2.500000000000000-3.500000000000000*I
sage: a.mpmath_to_sage(a.mpf('inf'), 53)
+infinity
sage: a.mpmath_to_sage(a.mpf('-inf'), 53)
-infinity
sage: a.mpmath_to_sage(a.mpf('nan'), 53)
NaN
sage: a.mpmath_to_sage(a.mpf('0'), 53)
0.000000000000000
```

A real example:

```python
sage: RealField(100)(pi)
3.1415926535897932384626433833
sage: t = RealField(100)(pi).mpmath_(); t
mpf('3.1415926535897932384626433833')
sage: a.mpmath_to_sage(t, 100)
3.1415926535897932384626433833
```

We can ask for more precision, but the result is undefined:
A complex example:

\begin{verbatim}
sage: ComplexField(100)([0, pi])
3.1415926535897932384626433833*I
sage: t = ComplexField(100)([0, pi])._mpmath_(); t
mpc(real='0.0', imag='3.1415926535897932')
sage: sage.libs.mpmath.all.mpmath_to_sage(t, 100)
3.1415926535897932384626433833*I
\end{verbatim}

Again, we can ask for more precision, but the result is undefined:

\begin{verbatim}
sage: sage.libs.mpmath.all.mpmath_to_sage(t, 140) # random
3.1415926535897932384626433833156440*I
sage: ComplexField(140)([0, pi])
3.1415926535897932384626433832795028841972*I
\end{verbatim}

\section*{normalize}

Create normalized mpf value tuple from full list of components.

\begin{verbatim}
sage: from mpmath.libmp import normalize
sage: normalize(0, 4, 5, 3, 53, 'n')
(0, 1, 7, 1)
\end{verbatim}

\section*{sage_to_mpmath}

Convert any Sage number that can be coerced into a RealNumber or ComplexNumber of the given precision into an mpmath mpf or mpc. Integers are currently converted to int.

Lists, tuples and dicts passed as input are converted recursively.

\begin{verbatim}
sage: import sage.libs.mpmath.all as a
sage: a.mp.dps = 15
sage: print(a.sage_to_mpmath(2/3, 53))
0.666666666666667
sage: print(a.sage_to_mpmath(2./3, 53))
0.666666666666667
sage: print(a.sage_to_mpmath(3+4*I, 53))
(3.0 + 4.0j)
sage: print(a.sage_to_mpmath(1+pi, 53))
4.14159265358979
sage: a.sage_to_mpmath(infinity, 53)
mpf('+inf')
sage: a.sage_to_mpmath(-infinity, 53)
mpf('-inf')
sage: a.sage_to_mpmath(NaN, 53)
mpf('nan')
sage: a.sage_to_mpmath(0, 53)
0
sage: a.sage_to_mpmath([0.5, 1.5], 53)
[mpf('0.5'), mpf('1.5')]
\end{verbatim}
sage: a.sage_to_mpmath((0.5, 1.5), 53)
(mpf('0.5'), mpf('1.5'))
sage: a.sage_to_mpmath({'n':0.5}, 53)
{'n': mpf('0.5')}
12.1 Victor Shoup's NTL C++ Library

Sage provides an interface to Victor Shoup’s C++ library NTL. Features of this library include incredibly fast arithmetic with polynomials and asymptotically fast factorization of polynomials.
13.1 Interface between Sage and PARI

13.1.1 Guide to real precision in the PARI interface

In the PARI interface, “real precision” refers to the precision of real numbers, so it is the floating-point precision. This is a non-trivial issue, since there are various interfaces for different things.

Internal representation and conversion between Sage and PARI

Real numbers in PARI have a precision associated to them, which is always a multiple of the CPU wordsize. So, it is a multiple of 32 or 64 bits. When converting from Sage to PARI, the precision is rounded up to the nearest multiple of the wordsize:

```sage
x = 1.0
x = x.precision()
53
pari(x)
1.00000000000000
pari(x).bitprecision()
64
```

With a higher precision:

```sage
x = RealField(100).pi()
x = x.precision()
100
pari(x).bitprecision()
128
```

When converting back to Sage, the precision from PARI is taken:

```sage
x = RealField(100).pi()
y = pari(x).sage()
y
3.1415926535897932384626433832793333156
parent(y)
Real Field with 128 bits of precision
```

So `pari(x).sage()` is definitely not equal to `x` since it has 28 bogus bits.

Therefore, some care must be taken when juggling reals back and forth between Sage and PARI. The correct way of avoiding this is to convert `pari(x).sage()` back into a domain with the right precision. This has to be done by
the user (or by Sage functions that use PARI library functions). For instance, if we want to use the PARI library to compute $\sqrt{\pi}$ with a precision of 100 bits:

```sage
definitions =
sage: R = RealField(100)
sage: s = R(pi); s
3.1415926535897932384626433833
sage: p = pari(s).sqrt()
sage: x = p.sage(); x # wow, more digits than I expected!
1.7724538509055160272981674833410973484
sage: x.prec() # has precision 'improved' from 100 to 128?
128
sage: x == RealField(128)(pi).sqrt() # sadly, no!
False
sage: R(x) # x should be brought back to precision 100
1.7724538509055160272981674833
sage: R(x) == s.sqrt() True
```

### Output precision for printing

Even though PARI reals have a precision, not all significant bits are printed by default. The maximum number of digits when printing a PARI real can be set using the methods `Pari.set_real_precision_bits()` or `Pari.set_real_precision()`.

We create a very precise approximation of $\pi$ and see how it is printed in PARI:

```sage
definitions =
sage: pi = pari(RealField(1000).pi())
sage: _.prec() # has precision 'improved' from 100 to 128?
128
sage: pi # x should be brought back to precision 100
3.14159265358979323846264338334109734833
sage: R(x) == s.sqrt() True
```

### Input precision for function calls

When we talk about precision for PARI functions, we need to distinguish three kinds of calls:

1. Using the string interface, for example `pari("\sin(1)")`.
2. Using the library interface with exact inputs, for example `pari(1).sin()`.
3. Using the library interface with inexact inputs, for example `pari(1.0).sin()`.

In the first case, the relevant precision is the one set by the methods `Pari.set_real_precision_bits()` or `Pari.set_real_precision()`.
In the second case, the precision can be given as the argument `precision` in the function call, with a default of 53 bits. The real precision set by `Pari.set_real_precision_bits()` or `Pari.set_real_precision()` is irrelevant.

In these examples, we convert to Sage to ensure that PARI’s real precision is not used when printing the numbers. As explained before, this artificially increases the precision to a multiple of the wordsize.

```python
s = pari(1).sin(precision=180).sage(); print(s); print(parent(s))
0.841470984807896507
Real Field with 192 bits of precision
s = pari(1).sin(precision=40).sage(); print(s); print(parent(s))
0.84147098407896507
Real Field with 64 bits of precision
s = pari(1).sin().sage(); print(s); print(parent(s))
0.84147098407896507
Real Field with 64 bits of precision
```

In the third case, the precision is determined only by the inexact inputs and the `precision` argument is ignored:

```python
s = pari(1.0).sin(precision=180).sage()
0.84147098407896507
s = pari(1.0).sin(precision=40).sage()
0.84147098407896507
s = pari(RealField(100).one()).sin().sage()
0.84147098407896507
```

### Elliptic curve functions

An elliptic curve given with exact \( a \)-invariants is considered an exact object. Therefore, you should set the precision for each method call individually:

```python
e = pari([0,0,0,-82,0]).ellinit()
sage: etal = e.elleta(precision=100)[0]
sage: etal.sage() 3.6054636014326520859158205642077267748
sage: etal = e.elleta(precision=180)[0]
sage: etal.sage() 3.605463601432652085915820564207726774810268999659802474544
```

### 13.2 Convert PARI objects to Sage types

`sage.libs.pari.convert_sage.gen_to_sage(z, locals=None)`

Convert a PARI gen to a Sage/Python object.

**INPUT:**

- \( z \) – PARI gen
• **locals** – optional dictionary used in fallback cases that involve `sage_eval()`

**OUTPUT:**

One of the following depending on the PARI type of `z`

- a **Integer** if `z` is an integer (type `t_INT`)
- a **Rational** if `z` is a rational (type `t_FRAC`)
- a **RealNumber** if `z` is a real number (type `t_REAL`). The precision will be equivalent.
- a **NumberFieldElement_quadratic** or a **ComplexNumber** if `z` is a complex number (type `t_COMPLEX`). The former is used when the real and imaginary parts are integers or rationals and the latter when they are floating point numbers. In that case the precision will be the maximal precision of the real and imaginary parts.
- a Python list if `z` is a vector or a list (type `t_VEC`, `t_COL`)
- a Python string if `z` is a string (type `t_STR`)
- a Python list of Python integers if `z` is a small vector (type `t_VECSMALL`)
- a matrix if `z` is a matrix (type `t_MAT`)
- a padic element (type `t_PADIC`)
- a **Infinity** if `z` is an infinity (type `t_INF`)

**EXAMPLES:**

```sage
from sage.libs.pari.convert_sage import gen_to_sage

Converting an integer:
```
sage: z = pari('12'); z
12
sage: z.type()
't_INT'
sage: a = gen_to_sage(z); a
12
sage: a.parent()
Integer Ring
```

```sage
sage: gen_to_sage(pari('7^42'))
31197348228452371301330321821976049
```

Converting a rational number:

```sage
sage: z = pari('389/17'); z
389/17
sage: z.type()
't_FRAC'
sage: a = gen_to_sage(z); a
389/17
sage: a.parent()
Rational Field
```

```sage
sage: gen_to_sage(pari('5^30 / 3^50'))
931322574615478515625/717897987691852588770249
```

Converting a real number:
pari.set_real_precision(70)
15
sage: z = pari('1.234'); z
1.234000000000000000000000000000000000000000000000000000000000000000000
sage: a = gen_to_sage(z); a
1.234000000000000000000000000000000000000000000000000000000000000000000000000
sage: a.parent()
Real Field with 256 bits of precision
sage: pari.set_real_precision(15)
70
sage: a = gen_to_sage(pari('1.234')); a
1.23400000000000000
sage: a.parent()
Real Field with 64 bits of precision

For complex numbers, the parent depends on the PARI type:

sage: z = pari('(3+I)'); z
3 + I
sage: z.type()
't_COMPLEX'
sage: a = gen_to_sage(z); a
i + 3
sage: a.parent()
Number Field in i with defining polynomial x^2 + 1

sage: z = pari('(3+I)/2'); z
3/2 + 1/2*I
sage: a = gen_to_sage(z); a
1/2*i + 3/2
sage: a.parent()
Number Field in i with defining polynomial x^2 + 1

sage: z = pari('1.0 + 2.0*I'); z
1.00000000000000 + 2.00000000000000*I
sage: a = gen_to_sage(z); a
1.00000000000000000 + 2.00000000000000000*I
sage: a.parent()
Complex Field with 64 bits of precision

Converting polynomials:

sage: f = pari('(2/3)*x^3 + x - 5/7 + y')
sage: f.type()
't_POL'
sage: R.<x,y> = QQ[]
sage: gen_to_sage(f, {'x': x, 'y': y})
2/3*x^3 + x + y - 5/7
sage: parent(gen_to_sage(f, {'x': x, 'y': y}))
Multivariate Polynomial Ring in x, y over Rational Field
sage: x,y = SR.var('x,y')
sage: gen_to_sage(f, {'x': x, 'y': y})
2/3*x^3 + x + y - 5/7
sage: parent(gen_to_sage(f, {'x': x, 'y': y}))
Symbolic Ring

(continues on next page)
Converting vectors:

```plaintext
sage: z1 = pari('[[-3, 2.1, 1+I]]; z1
[-3, 2.10000000000000, 1 + I]
sage: z2 = pari('[1.0+I, [1,2]~]); z2
[1.00000000000000+I, [1, 2]~]
sage: z1.type(), z2.type()
('t_VEC', 't_COL')
sage: a1 = gen_to_sage(z1)
sage: a2 = gen_to_sage(z2)
sage: type(a1), type(a2)
(<... 'list'>, <... 'list'>)
sage: [parent(b) for b in a1]
[Integer Ring,
  Real Field with 64 bits of precision,
  Number Field in i with defining polynomial x^2 + 1]
sage: [parent(b) for b in a2]
[Complex Field with 64 bits of precision, <... 'list'>]
sage: z = pari('Vecsmall([1,2,3,4])')
sage: z.type()
't_VECSMALL'
sage: a = gen_to_sage(z); a
[1, 2, 3, 4]
sage: a.parent()
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
```

Matrices:

```plaintext
sage: z = pari('[1,2;3,4]')
sage: z.type()
't_MAT'
sage: a = gen_to_sage(z); a
[1 2]
[3 4]
sage: a.parent()
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
```

Conversion of p-adics:

```plaintext
sage: z = pari('569 + O(7^8)'); z
2 + 4*7 + 4*7^2 + 7^3 + O(7^8)
sage: a = gen_to_sage(z); a
2 + 4*7 + 4*7^2 + 7^3 + O(7^8)
sage: a.parent()
7-adic Field with capped relative precision 8
```

Conversion of infinities:

```plaintext
```

(continued from previous page)
Conversion of strings:

```
sage: s = pari("foo").sage(); s
'foo'
sage: type(s)
<type 'str'>
```

## 13.3 Ring of pari objects

AUTHORS:

- Simon King (2011-08-24): Use UniqueRepresentation, \texttt{element}\_\texttt{class} and proper initialisation of elements.

```python
class sage.rings.pari_ring.Pari(x, parent=None)
    Bases: sage.structure.element.RingElement
    Element of Pari pseudo-ring.

class sage.rings.pari_ring.PariRing
    Bases: sage.misc.fast_methods.Singleton, sage.rings.ring.Ring
    EXAMPLES:
```

```
sage: R = PariRing(); R
Pseudoring of all PARI objects.
sage: loads(R.dumps()) is R
True
```

**Element**

alias of \texttt{Pari}

**characteristic()**

**is\_field**(\texttt{proof=True})

**random\_element**(\texttt{x=None}, \texttt{y=None}, \texttt{distribution=None})

Return a random integer in Pari.

**Note:** The given arguments are passed to \texttt{ZZ.random_element(...)}.

**INPUT:**

- \texttt{x, y} – optional integers, that are lower and upper bound for the result. If only \texttt{x} is provided, then the result is between 0 and \texttt{x} – 1, inclusive. If both are provided, then the result is between \texttt{x} and \texttt{y} – 1, inclusive.
- \texttt{distribution} – optional string, so that \texttt{ZZ} can make sense of it as a probability distribution.

**EXAMPLES:**
sage: R = PariRing()
sage: R.random_element()
-8
sage: R.random_element(5,13)
12
sage: [R.random_element(distribution="1/n") for _ in range(10)]
[0, 1, -1, 2, 1, -95, -1, -2, -12, 0]

\texttt{zeta()} \hfill \\
Return -1.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: R = PariRing()
sage: R.zeta()
-1
\end{verbatim}
14.1 Cython wrapper for the Parma Polyhedra Library (PPL)

The Parma Polyhedra Library (PPL) is a library for polyhedral computations over $\mathbb{Q}$. This interface tries to reproduce the C++ API as faithfully as possible in Cython/Sage. For example, the following C++ excerpt:

```cpp
Variable x(0);
Variable y(1);
Constraint_System cs;
cs.insert(x >= 0);
cs.insert(x <= 3);
cs.insert(y >= 0);
cs.insert(y <= 3);
C_Polyhedron poly_from_constraints(cs);
```

translates into:

```python
sage: from sage.libs.ppl import Variable, Constraint_System, C_Polyhedron
doctest:warning
...
DeprecationWarning: The Sage wrappers around ppl are now superseded by the standalone pplpy.
Please use import 'ppl' instead of 'sage.libs.ppl'.
See http://trac.sagemath.org/23024 for details.
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert(x >= 0)
sage: cs.insert(x <= 3)
sage: cs.insert(y >= 0)
sage: cs.insert(y <= 3)
sage: poly_from_constraints = C_Polyhedron(cs)
```

The same polyhedron constructed from generators:

```python
sage: from sage.libs.ppl import Variable, Generator_System, C_Polyhedron, point
sage: gs = Generator_System()
sage: gs.insert(point(0*x + 0*y))
sage: gs.insert(point(0*x + 3*y))
sage: gs.insert(point(3*x + 0*y))
sage: gs.insert(point(3*x + 3*y))
sage: poly_from_generators = C_Polyhedron(gs)
```

Rich comparisons test equality/inequality and strict/non-strict containment:
As we see above, the library is generally easy to use. There are a few pitfalls that are not entirely obvious without consulting the documentation, in particular:

- There are no vectors used to describe `Generator` (points, closure points, rays, lines) or `Constraint` (strict inequalities, non-strict inequalities, or equations). Coordinates are always specified via linear polynomials in `Variable`
- All coordinates of rays and lines as well as all coefficients of constraint relations are (arbitrary precision) integers. Only the generators `point()` and `closure_point()` allow one to specify an overall divisor of the otherwise integral coordinates. For example:

```python
sage: from sage.libs.ppl import Variable, point
sage: x = Variable(0); y = Variable(1)

sage: p = point(2*x+3*y, 5); p
point(2/5, 3/5)
sage: p.coefficient(x)
2
sage: p.coefficient(y)
3
sage: p.divisor()
5
```

- PPL supports (topologically) closed polyhedra (`C_Polyhedron`) as well as not necessarily closed polyhedra (`NNC_Polyhedron`). Only the latter allows closure points (=points of the closure but not of the actual polyhedron) and strict inequalities (`>` and `<`)

The naming convention for the C++ classes is that they start with `PPL_`, for example, the original `Linear_Expression` becomes `PPL_Linear_Expression`. The Python wrapper has the same name as the original library class, that is, just `Linear_Expression`. In short:

- If you are using the Python wrapper (if in doubt: thats you), then you use the same names as the PPL C++ class library.
- If you are writing your own Cython code, you can access the underlying C++ classes by adding the prefix `PPL_`

Finally, PPL is fast. For example, here is the permutahedron of 5 basis vectors:

```python
sage: from sage.libs.ppl import Variable, Generator_System, point, C_Polyhedron
sage: basis = list(range(5))
sage: x = [ Variable(i) for i in basis ]

sage: gs = Generator_System()

sage: for coeff in Permutations(basis):
....:    gs.insert(point( sum( (coeff[i]+1)*x[i] for i in basis ) ))

sage: C_Polyhedron(gs)
A 4-dimensional polyhedron in QQ^5 defined as the convex hull of 120 points
```

The same computation with cddlib which is slightly slower:
```python
sage: basis = list(range(5))
sage: gs = [ tuple(coeff) for coeff in Permutations(basis) ]
sage: Polyhedron(vertices=gs, backend='cdd')
```

A 4-dimensional polyhedron in $\mathbb{Q}^5$ defined as the convex hull of 120 vertices

**DIFFERENCES VS. C++**

Since Python and C++ syntax are not always compatible, there are necessarily some differences. The main ones are:

- The `Linear_Expression` also accepts an iterable as input for the homogeneous coefficients.
- `Polyhedron` and its subclasses as well as `Generator_System` and `Constraint_System` can be set immutable via a `set_immutable()` method. This is the analog of declaring a C++ instance `const`. All other classes are immutable by themselves.

**AUTHORS:**

- Volker Braun (2010-10-08): initial version.
- Risan (2012-02-19): extension for MIP_Problem class
- Vincent Klein (2017-12-21): Deprecate this module. Future change should be done in the standalone package pplpy (https://github.com/videlec/pplpy).

```
class sage.libs.ppl.C_Polyhedron
    Bases: sage.libs.ppl.Polyhedron

    Wrapper for PPL's C_Polyhedron class.

    An object of the class `C_Polyhedron` represents a topologically closed convex polyhedron in the vector space. See `NNC_Polyhedron` for more general (not necessarily closed) polyhedra.

    When building a closed polyhedron starting from a system of constraints, an exception is thrown if the system contains a strict inequality constraint. Similarly, an exception is thrown when building a closed polyhedron starting from a system of generators containing a closure point.

    INPUT:

    - `arg` – the defining data of the polyhedron. Any one of the following is accepted:
      - A non-negative integer. Depending on `degenerate_element`, either the space-filling or the empty polytope in the given dimension `arg` is constructed.
      - A `Constraint_System`.
      - A `Generator_System`.
      - A single `Constraint`.
      - A single `Generator`.
    - A `C_Polyhedron`.
    - `degenerate_element` – string, either 'universe' or 'empty'. Only used if `arg` is an integer.

    OUTPUT:

    A `C_Polyhedron`.

    EXAMPLES:
```
```
The empty and universe polyhedra are constructed like this:

```python
sage: C_Polyhedron(3, 'empty')
The empty polyhedron in QQ^3
sage: C_Polyhedron(3, 'empty').constraints()
Constraint_System {-1==0}
```

Note that, by convention, the generator system of a polyhedron is either empty or contains at least one point. In particular, if you define a polyhedron via a non-empty `Generator_System` it must contain a point (at any position). If you start with a single generator, this generator must be a point:

```python
sage: C_Polyhedron( ray(x) )
Traceback (most recent call last):
  ...
ValueError: PPL::C_Polyhedron::C_Polyhedron(gs):
*this is an empty polyhedron and
the non-empty generator system gs contains no points.
```

### class `sage.libs.ppl.Constraint`

**Bases:** object

Wrapper for PPL's `Constraint` class.

An object of the class `Constraint` is either:

- an equality \( \sum_{i=0}^{n-1} a_i x_i + b = 0 \)
- a non-strict inequality \( \sum_{i=0}^{n-1} a_i x_i + b \geq 0 \)
- a strict inequality \( \sum_{i=0}^{n-1} a_i x_i + b > 0 \)

where \( n \) is the dimension of the space, \( a_i \) is the integer coefficient of variable \( x_i \), and \( b_i \) is the integer inhomogeneous term.

**INPUT/OUTPUT:**

You construct constraints by writing inequalities in `LinearExpression`. Do not attempt to manually construct constraints.

**EXAMPLES:**
Sage Reference Manual: C/C++ Library Interfaces, Release 8.7

```python
sage: from sage.libs.ppl import Constraint, Variable, Linear_Expression
sage: x = Variable(0)
sage: y = Variable(1)
sage: 5*x-2*y > x+y-1
4*x0-3*x1+1>0
sage: 5*x-2*y >= x+y-1
4*x0-3*x1+1>=0
sage: 5*x-2*y == x+y-1
4*x0-3*x1+1==0
sage: 5*x-2*y <= x+y-1
-4*x0+3*x1-1>=0
sage: 5*x-2*y < x+y-1
-4*x0+3*x1-1>0
sage: x > 0
x0>0
```

Special care is needed if the left hand side is a constant:

```python
sage: 0 == 1      # watch out!
False
sage: Linear_Expression(0) == 1
-1==0
```

**OK()**

Check if all the invariants are satisfied.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Linear_Expression, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: ineq = (3*x+2*y+1>=0)
sage: ineq.OK()
True
```

**ascii_dump()**

Write an ASCII dump to stderr.

**EXAMPLES:**

```python
sage: sage_cmd = 'from sage.libs.ppl import Linear_Expression, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: e = (3*x+2*y+1 > 0)
sage: e.ascii_dump()

sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd],
˓→timeout=100) # long time, indirect doctest
sage: print(err)
//... DeprecationWarning: The Sage wrappers around ppl are now superseded by
˓→the standalone pplpy.
Please use import 'ppl' instead of 'sage.libs.ppl'.
See http://trac.sagemath.org/23024 for details.
... size 4 1 3 2 -1 > (NNC)
```

**coefficient(v)**

Return the coefficient of the variable v.

14.1. Cython wrapper for the Parma Polyhedra Library (PPL)
INPUT:

- \( v \) – a `Variable`.

OUTPUT:

An integer.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: ineq = (3*x+1 > 0)
sage: ineq.coefficient(x)
3
```

`coefficients()`

Return the coefficients of the constraint.

See also `coefficient()`.

OUTPUT:

A tuple of integers of length `space_dimension()`.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable
sage: x = Variable(0); y = Variable(1)
sage: ineq = (3*x+5*y+1 == 2); ineq
3*x0+5*x1-1==0
sage: ineq.coefficients()
(3, 5)
```

`inhomogeneous_term()`

Return the inhomogeneous term of the constraint.

OUTPUT:

Integer.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable
sage: y = Variable(1)
sage: ineq = (10+y > 9)
sage: ineq
x1+1>0
sage: ineq.inhomogeneous_term()
1
```

`is_equality()`

Test whether `self` is an equality.

OUTPUT:

Boolean. Returns `True` if and only if `self` is an equality constraint.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
(continues on next page)```
True
sage: (x>=0).is_equality()
False
sage: (x>0).is_equality()
False

**is_equivalent_to**\((c)\)
Test whether *self* and *c* are equivalent.

**INPUT:**
- *c* – a *Constraint*.

**OUTPUT:**
Boolean. Returns True if and only if *self* and *c* are equivalent constraints.

Note that constraints having different space dimensions are not equivalent. However, constraints having different types may nonetheless be equivalent, if they both are tautologies or inconsistent.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Linear_Expression
sage: x = Variable(0)
sage: y = Variable(1)
sage: (x>0).is_equivalent_to(Linear_Expression(0)<x)
True
sage: (x>0).is_equivalent_to(0*y<x)
False
sage: (0*x>1).is_equivalent_to(0*x==-2)
True
```

**is_inconsistent**()
Test whether *self* is an inconsistent constraint, that is, always false.

An inconsistent constraint can have either one of the following forms:
- an equality: \(\sum 0x_i + b = 0\) with \(b \neq 0\),
- a non-strict inequality: \(\sum 0x_i + b \geq 0\) with \(b < 0\), or
- a strict inequality: \(\sum 0x_i + b > 0\) with \(b \leq 0\).

**OUTPUT:**
Boolean. Returns True if and only if *self* is an inconsistent constraint.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: (x==1).is_inconsistent()
False
sage: (0*x>=1).is_inconsistent()
True
```

**is_inequality**()
Test whether *self* is an inequality.

**OUTPUT:**
Boolean. Returns True if and only if *self* is an inequality constraint, either strict or non-strict.
EXAMPLES:

```python
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
```

```python
sage: (x==0).is_inequality()
False
```

```python
sage: (x>=0).is_inequality()
True
```

```python
sage: (x>0).is_inequality()
True
```

**is_nonstrict_inequality()**

Test whether `self` is a non-strict inequality.

**OUTPUT:**

Boolean. Returns `True` if and only if `self` is a non-strict inequality constraint.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
```

```python
sage: (x==0).is_nonstrict_inequality()
False
```

```python
sage: (x>=0).is_nonstrict_inequality()
True
```

```python
sage: (x>0).is_nonstrict_inequality()
False
```

**is_strict_inequality()**

Test whether `self` is a strict inequality.

**OUTPUT:**

Boolean. Returns `True` if and only if `self` is a strict inequality constraint.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
```

```python
sage: (x==0).is_strict_inequality()
False
```

```python
sage: (x>=0).is_strict_inequality()
False
```

```python
sage: (x>0).is_strict_inequality()
True
```

**is_tautological()**

Test whether `self` is a tautological constraint.

A tautology can have either one of the following forms:

- an equality: $\sum_0 x_i + 0 = 0$,
- a non-strict inequality: $\sum_0 x_i + b \geq 0$ with $b \geq 0$, or
- a strict inequality: $\sum_0 x_i + b > 0$ with $b > 0$.

**OUTPUT:**

Boolean. Returns `True` if and only if `self` is a tautological constraint.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: (x==0).is_tautological()
False
sage: (0*x>=0).is_tautological()
True

**space_dimension()**

Return the dimension of the vector space enclosing `self`.

**OUTPUT:**

Integer.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: (x>=0).space_dimension()
1
sage: (y==1).space_dimension()
2
```

**type()**

Return the constraint type of `self`.

**OUTPUT:**

String. One of 'equality', 'nonstrict_inequality', or 'strict_inequality'.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: (x==0).type()
'equality'
sage: (x>=0).type()
'nonstrict_inequality'
sage: (x>0).type()
'strict_inequality'
```

**class** `sage.libs.ppl.Constraint_System`

*Bases:.* `sage.libs.ppl._mutable_or_immutable`

Wrapper for PPL's `Constraint_System` class.

An object of the class `Constraint_System` is a system of constraints, i.e., a multiset of objects of the class `Constraint`. When inserting constraints in a system, space dimensions are automatically adjusted so that all the constraints in the system are defined on the same vector space.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Constraint_System, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System( 5*x-2*y > 0 )
sage: cs.insert( 6*x<3*y )
sage: cs.insert( x >= 2*x-7*y )
```

(continues on next page)
sage: cs
Constraint_System (5*x0-2*x1>0, -2*x0+x1>0, -x0+7*x1>=0)

OK()

Check if all the invariants are satisfied.

EXAMPLES:

sage: from sage.libs.ppl import Variable, Constraint_System
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System( 3*x+2*y+1 <= 10 )
sage: cs.OK()
True

ascii_dump()

Write an ASCII dump to stderr.

EXAMPLES:

sage: sage_cmd = 'from sage.libs.ppl import Constraint_System, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System( 3*x > 2*y+1 )
sage: cs.ascii_dump()

sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd],
timeout=100)  # long time, indirect doctest
sage: print(err)
# long time
/... DeprecationWarning: The Sage wrappers around ppl are now superseded by
the standalone pplpy.
Please use import 'ppl' instead of 'sage.libs.ppl'.
...
  topology NOT_NECESSARILY_CLOSED
  1 x 2 SPARSE (sorted)
  index_first_pending 1
  size 4 -1 3 -2 -1 > (NNC)

clear()

Removes all constraints from the constraint system and sets its space dimension to 0.

EXAMPLES:

sage: from sage.libs.ppl import Variable, Constraint_System
sage: x = Variable(0)
sage: cs = Constraint_System(x>0)
sage: cs
Constraint_System {x0>0}
sage: cs.clear()
sage: cs
Constraint_System {}

empty()

Return True if and only if self has no constraints.

OUTPUT:

Boolean.
EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, Constraint_System, point
sage: x = Variable(0)
sage: cs = Constraint_System()
sage: cs.empty()
    True
sage: cs.insert( x>0 )
sage: cs.empty()
    False
```

`has_equalities()`
Tests whether `self` contains one or more equality constraints.

OUTPUT:
Boolean.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, Constraint_System
sage: x = Variable(0)
sage: cs = Constraint_System()
sage: cs.insert( x>0 )
sage: cs.insert( x<0 )
sage: cs.has_equalities()
    False
sage: cs.insert( x==0 )
sage: cs.has_equalities()
    True
```

`has_strict_inequalities()`
Tests whether `self` contains one or more strict inequality constraints.

OUTPUT:
Boolean.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, Constraint_System
sage: x = Variable(0)
sage: cs = Constraint_System()
sage: cs.insert( x>=0 )
sage: cs.insert( x==-1 )
sage: cs.has_strict_inequalities()
    False
sage: cs.insert( x>0 )
sage: cs.has_strict_inequalities()
    True
```

`insert(c)`
Insert `c` into the constraint system.

INPUT:

- `c` — a `Constraint`.

EXAMPLES:
```
sage: from sage.libs.ppl import Variable, Constraint_System
sage: x = Variable(0)
sage: cs = Constraint_System()
sage: cs.insert( x>0 )
sage: cs
Constraint_System {x0>0}
```

**space_dimension()**
Return the dimension of the vector space enclosing `self`.

**OUTPUT:**
Integer.

**EXAMPLES:**
```
sage: from sage.libs.ppl import Variable, Constraint_System
sage: x = Variable(0)
sage: cs = Constraint_System( x>0 )
sage: cs.space_dimension()
1
```

**class sage.libs.ppl.Constraint_System_iterator**
Bases: object
Wrapper for PPL's `Constraint_System::const_iterator` class.

**EXAMPLES:**
```
sage: from sage.libs.ppl import Constraint_System, Variable, Constraint_System_iterator
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System( 5*x < 2*y )
sage: cs.insert( 6*x-3*y==0 )
sage: cs.insert( x >= 2*x-7*y )
sage: next(Constraint_System_iterator(cs))
-5*x0+2*x1>0
sage: list(cs)
[-5*x0+2*x1>0, 2*x0-x1==0, -x0+7*x1>0]
```

**next()**
`x.next()` -> the next value, or raise StopIteration

**class sage.libs.ppl.Generator**
Bases: object
Wrapper for PPL's `Generator` class.

An object of the class `Generator` is one of the following:
- a line $\ell = (a_0, \ldots, a_{n-1})^T$
- a ray $r = (a_0, \ldots, a_{n-1})^T$
- a point $p = \left(\frac{a_0}{d}, \ldots, \frac{a_{n-1}}{d}\right)^T$
- a closure point $c = \left(\frac{a_0}{d}, \ldots, \frac{a_{n-1}}{d}\right)^T$

where $n$ is the dimension of the space and, for points and closure points, $d$ is the divisor.

**INPUT/OUTPUT:**

138 Chapter 14. PPL
Use the helper functions \texttt{line()}, \texttt{ray()}, \texttt{point()}, and \texttt{closure\_point()} to construct generators. Analogous class methods are also available, see \texttt{Generator.line()}, \texttt{Generator.ray()}, \texttt{Generator.point()}, \texttt{Generator.closure\_point()}. Do not attempt to construct generators manually.

\textbf{Note:} The generators are constructed from linear expressions. The inhomogeneous term is always silently discarded.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.libs.ppl import Generator, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: Generator.line(5*x-2*y)
line(5, -2)
sage: Generator.ray(5*x-2*y)
ray(5, -2)
sage: Generator.point(5*x-2*y, 7)
point(5/7, -2/7)
sage: Generator.closure_point(5*x-2*y, 7)
closure_point(5/7, -2/7)
\end{verbatim}

\texttt{OK()}
Check if all the invariants are satisfied.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.libs.ppl import Linear\_Expression, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: e = 3*x+2*y+1
sage: e.OK()
True
\end{verbatim}

\texttt{ascii\_dump()}
Write an ASCII dump to stderr.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: sage\_cmd = 'from sage.libs.ppl import Linear\_Expression, Variable, ' +
\hspace{1cm} 'ppl' +
\hspace{1cm} 'point'
\end{verbatim}

\begin{verbatim}
sage: sage\_cmd += 'x = Variable(0)\n'
sage: sage\_cmd += 'y = Variable(1)\n'
sage: sage\_cmd += 'p = point(3\times+2\times)\n'
sage: sage\_cmd += 'p.ascii\_dump()\n'
\end{verbatim}

\begin{verbatim}
sage: from sage.tests.cmdline import test\_executable
sage: (out, err, ret) = test\_executable(['sage', '-c', sage\_cmd],
\hspace{1cm} 'timeout=100') # long time, indirect doctest
sage: print(err)
# long time
/... DeprecationWarning: The Sage wrappers around ppl are now superseded by the standalone pplpy.
Please use import 'ppl' instead of 'sage.libs.ppl'. See http://trac.sagemath.org/23024 for details.
...
size 3 1 3 2 P (C)
\end{verbatim}

\texttt{closure\_point(\texttt{expression}=0, \texttt{divisor}=1)}
Construct a closure point.
A closure point is a point of the topological closure of a polyhedron that is not a point of the polyhedron itself.

**INPUT:**

- expression – a Linear Expression or something convertible to it (Variable or integer).
- divisor – an integer.

**OUTPUT:**

A new Generator representing the point.

Raises a ValueError if `divisor==0`.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Generator, Variable
sage: y = Variable(1)
```

```python
sage: Generator.closure_point(2*y+7, 3)
```

```python
closure_point(0/3, 2/3)
```

```python
sage: Generator.closure_point(y+7, 3)
```

```python
closure_point(0/3, 1/3)
```

```python
sage: Generator.closure_point(7, 3)
```

```python
closure_point()
```

```python
sage: Generator.closure_point(0, 0)
```

Traceback (most recent call last):
...

`ValueError: PPL::closure_point(e, d): d == 0.`

`coefficient (v)`

Return the coefficient of the variable v.

**INPUT:**

- v – a Variable.

**OUTPUT:**

An integer.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, line
sage: x = Variable(0)
```

```python
sage: line = line(3*x+1)
```

```python
sage: line
```

```python
line(1)
```

```python
sage: line.coefficient(x)
```

```python
1
```

`coefficients ()`

Return the coefficients of the generator.

See also `coefficient ()`.

**OUTPUT:**

A tuple of integers of length `space_dimension ()`.

**EXAMPLES:**
```
sage: from sage.libs.ppl import Variable, point
sage: x = Variable(0); y = Variable(1)
sage: p = point(3*x+5*y+1, 2); p
point(3/2, 5/2)
sage: p.coefficients()
(3, 5)
```

**divisor()**

If `self` is either a point or a closure point, return its divisor.

**OUTPUT:**

An integer. If `self` is a ray or a line, raises `ValueError`.

**EXAMPLES:**

```
sage: from sage.libs.ppl import Generator, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: point = Generator.point(2*x-y+5)
sage: point.divisor()
1
sage: line = Generator.line(2*x-y+5)
sage: line.divisor()
Traceback (most recent call last):
...
ValueError: PPL::Generator::divisor():
  *this is neither a point nor a closure point.
```

**is_closure_point()**

Test whether `self` is a closure point.

**OUTPUT:**

Boolean.

**EXAMPLES:**

```
sage: from sage.libs.ppl import Variable, point, closure_point, ray, line
sage: x = Variable(0)
sage: line(x).is_closure_point()
False
sage: ray(x).is_closure_point()
False
sage: point(x,2).is_closure_point()
False
sage: closure_point(x,2).is_closure_point()
True
```

**is_equivalent_to(g)**

Test whether `self` and `g` are equivalent.

**INPUT:**

- `g` – a `Generator`.

**OUTPUT:**

Boolean. Returns `True` if and only if `self` and `g` are equivalent generators.

Note that generators having different space dimensions are not equivalent.
EXAMPLES:

```python
sage: from sage.libs.ppl import Generator, Variable, point, line
sage: x = Variable(0)
sage: y = Variable(1)
sage: point(2*x , 2).is_equivalent_to( point(x) )
True
sage: point(2*x+0*y, 2).is_equivalent_to( point(x) )
False
sage: line(4*x).is_equivalent_to(line(x))
True
```

**is_line()**

Test whether `self` is a line.

**OUTPUT:**

Boolean.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, point, closure_point, ray, line
sage: x = Variable(0)
sage: line(x).is_line()
True
sage: ray(x).is_line()
False
sage: point(x,2).is_line()
False
sage: closure_point(x,2).is_line()
False
```

**is_line_or_ray()**

Test whether `self` is a line or a ray.

**OUTPUT:**

Boolean.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, point, closure_point, ray, line
sage: x = Variable(0)
sage: line(x).is_line_or_ray()
True
sage: ray(x).is_line_or_ray()
True
sage: point(x,2).is_line_or_ray()
False
sage: closure_point(x,2).is_line_or_ray()
False
```

**is_point()**

Test whether `self` is a point.

**OUTPUT:**

Boolean.

**EXAMPLES:**
from sage.libs.ppl import Variable, point, closure_point, ray, line

sage: x = Variable(0)
sage: line(x).is_point()
False
sage: ray(x).is_point()
False
sage: point(x, 2).is_point()
True
sage: closure_point(x, 2).is_point()
False

is_ray()
Test whether self is a ray.

OUTPUT:
Boolean.

EXAMPLES:

sage: from sage.libs.ppl import Variable, point, closure_point, ray, line
sage: x = Variable(0)
sage: line(x).is_ray()
False
sage: ray(x).is_ray()
True
sage: point(x, 2).is_ray()
False
sage: closure_point(x, 2).is_ray()
False

line(expression)
Construct a line.

INPUT:

- expression— a Linear_Expression or something convertible to it (Variable or integer).

OUTPUT:
A new Generator representing the line.

Raises a ValueError if the homogeneous part of `expression represents the origin of the vector space.

EXAMPLES:

sage: from sage.libs.ppl import Generator, Variable
sage: y = Variable(1)
sage: Generator.line(2*y)
line(0, 1)
sage: Generator.line(y)
line(0, 1)
sage: Generator.line(1)
Traceback (most recent call last):
... ValueError: PPL::line(e):
e == 0, but the origin cannot be a line.

point(expression=0, divisor=1)
Construct a point.
INPUT:

- expression – a `Linear_Expression` or something convertible to it (`Variable` or integer).
- divisor – an integer.

OUTPUT:

A new `Generator` representing the point.

Raises a `ValueError` if `divisor==0`.

EXAMPLES:

```python
sage: from sage.libs.ppl import Generator, Variable
gle s: y = Variable(1)
sage: Generator.point(2*y, 3)
point(0/3, 2/3)
sage: Generator.point(y, 3)
point(0/3, 1/3)
sage: Generator.point(7, 3)
point()
sage: Generator.point(0, 0)
Traceback (most recent call last):
  ... ValueError: PPL::point(e, d):
    d == 0.
```

`ray(expression)`

Construct a ray.

INPUT:

- expression – a `Linear_Expression` or something convertible to it (`Variable` or integer).

OUTPUT:

A new `Generator` representing the ray.

Raises a `ValueError` if the homogeneous part of `expression` represents the origin of the vector space.

EXAMPLES:

```python
sage: from sage.libs.ppl import Generator, Variable
gle s: y = Variable(1)
sage: Generator.ray(2*y)
ray(0, 1)
sage: Generator.ray(y)
ray(0, 1)
sage: Generator.ray(1)
Traceback (most recent call last):
  ... ValueError: PPL::ray(e):
    e == 0, but the origin cannot be a ray.
```

`space_dimension()`

Return the dimension of the vector space enclosing `self`.

OUTPUT:

Integer.

EXAMPLES:
```python
sage: from sage.libs.ppl import Variable, point
sage: x = Variable(0)
sage: y = Variable(1)
sage: point(x).space_dimension()
1
sage: point(y).space_dimension()
2
```

### type()

Return the generator type of `self`.

**OUTPUT:**

String. One of 'line', 'ray', 'point', or 'closure_point'.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, line, ray, point, closure_point
sage: x = Variable(0)
sage: line(x).type()
'line'
sage: ray(x).type()
'ray'
sage: point(x,2).type()
'point'
sage: closure_point(x,2).type()
'closure_point'
```

### class sage.libs.ppl.Generator_System

**Bases:** `sage.libs.ppl._mutable_or_immutable`

Wrapper for PPL's `Generator_System` class.

An object of the class `Generator_System` is a system of generators, i.e., a multiset of objects of the class `Generator` (lines, rays, points and closure points). When inserting generators in a system, space dimensions are automatically adjusted so that all the generators in the system are defined on the same vector space. A system of generators which is meant to define a non-empty polyhedron must include at least one point: the reason is that lines, rays and closure points need a supporting point (lines and rays only specify directions while closure points only specify points in the topological closure of the NNC polyhedron).

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Generator_System, Variable, line, ray, point, closure_point
sage: x = Variable(0)
sage: y = Variable(1)
sage: gs = Generator_System( line(5*x-2*y) )
sage: gs.insert( ray(6*x-3*y) )
sage: gs.insert( point(2*x-7*y, 5) )
sage: gs.insert( closure_point(9*x-1*y, 2) )
sage: gs
Generator_System {line(5, -2), ray(2, -1), point(2/5, -7/5), closure_point(9/2, -1/2)}
```

### OK()

Check if all the invariants are satisfied.

**EXAMPLES:**

14.1. Cython wrapper for the Parma Polyhedra Library (PPL)
```python
sage: from sage.libs.ppl import Variable, Generator_System, point
sage: x = Variable(0)
sage: y = Variable(1)
sage: gs = Generator_System( point(3*x+2*y+1) )
sage: gs.OK()
True
```

**ascii_dump()**

Write an ASCII dump to stderr.

**EXAMPLES:**

```python
sage: sage_cmd = 'from sage.libs.ppl import Generator_System, point,
˓→Variable
sage: sage_cmd += 'x = Variable(0)\n'
sage: sage_cmd += 'y = Variable(1)\n'
sage: sage_cmd += 'gs = Generator_System( point(3*x+2*y+1) )\n'
sage: sage_cmd += 'gs.ascii_dump()\n'
sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd],
˓→timeout=100) # long time, indirect doctest
sage: print(err)
#/... DeprecationWarning: The Sage wrappers around ppl are now superseded by
˓→the standalone pplpy.
Please use import 'ppl' instead of 'sage.libs.ppl'.
See http://trac.sagemath.org/23024 for details.
...
topology NECESSARILY_CLOSED
1 x 2 SPARSE (sorted)
index_first_pending 1
size 3 1 3 2 P (C)
```

**clear()**

Removes all generators from the generator system and sets its space dimension to 0.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Generator_System, point
sage: x = Variable(0)
sage: gs = Generator_System() ; gs
Generator_System {
}
sage: gs.clear()
sage: gs
Generator_System {
}
```

**empty()**

Return True if and only if self has no generators.

**OUTPUT:**

Boolean.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Generator_System, point
sage: x = Variable(0)
sage: gs = Generator_System()
sage: gs.empty()
True
```
sage: gs.insert( point(-3*x) )
sage: gs.empty()
False

**insert**(*g*)

Insert *g* into the generator system.

The number of space dimensions of *self* is increased, if needed.

**INPUT:**

- *g* – a *Generator*.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Generator_System, point
sage: x = Variable(0)
sage: gs = Generator_System( point(3*x) )
sage: gs.insert( point(-3*x) )
sage: gs
Generator_System {point(3/1), point(-3/1)}
```

**space_dimension**()

Return the dimension of the vector space enclosing *self*.

**OUTPUT:**

Integer.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Generator_System, point
sage: x = Variable(0)
sage: gs = Generator_System( point(3*x) )
sage: gs.space_dimension()
1
```

### class sage.libs.ppl.Generator_System_iterator

**Bases:** object

Wrapper for PPL’s `Generator_System::const_iterator` class.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Generator_System, Variable, line, ray, point,
    closure_point, Generator_System_iterator
sage: x = Variable(0)
sage: y = Variable(1)
sage: gs = Generator_System( line(5*x-2*y) )
sage: gs.insert( ray(6*x-3*y) )
sage: gs.insert( point(2*x-7*y, 5) )
sage: gs.insert( closure_point(9*x-1*y, 9/2) )
sage: next(Generator_System_iterator(gs))
line(5, -2)
sage: list(gs)
[line(5, -2), ray(2, -1), point(2/5, -7/5), closure_point(9/2, -1/2)]
```

**next**()

`x.next()` -> the next value, or raise `StopIteration`
class sage.libs.ppl.Linear_Expression
Bases: object

Wrapper for PPL’s PPL_Linear_Expression class.

INPUT:
The constructor accepts zero, one, or two arguments.

If there are two arguments Linear_Expression(a, b), they are interpreted as
  • a – an iterable of integer coefficients, for example a list.
  • b – an integer. The inhomogeneous term.

A single argument Linear_Expression(arg) is interpreted as
  • arg – something that determines a linear expression. Possibilities are:
    – a Variable: The linear expression given by that variable.
    – a Linear_Expression: The copy constructor.
    – an integer: Constructs the constant linear expression.

No argument is the default constructor and returns the zero linear expression.

OUTPUT:
A Linear_Expression

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Linear_Expression
sage: Linear_Expression([1,2,3,4],5)
x0+2*x1+3*x2+4*x3+5
sage: Linear_Expression(10)
10
sage: Linear_Expression()
0
sage: Linear_Expression(10).inhomogeneous_term()
10
sage: x = Variable(123)
sage: expr = x+1; expr
x123+1
sage: expr.OK()
True
sage: expr.coefficient(x)
1
sage: expr.coefficient( Variable(124) )
0
```

OK ()
Check if all the invariants are satisfied.

EXAMPLES:

```
sage: from sage.libs.ppl import Linear_Expression, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: e = 3*x+2*y+1
sage: e.OK()
True
```
all_homogeneous_terms_are_zero()
Test if self is a constant linear expression.

OUTPUT:
Boolean.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, Linear_Expression
sage: Linear_Expression(10).all_homogeneous_terms_are_zero()
True
```

ascii_dump()
Write an ASCII dump to stderr.

EXAMPLES:

```python
sage: sage_cmd = 'from sage.libs.ppl import Linear_Expression, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: e = 3*x+2*y+1
sage: e.ascii_dump()

//... DeprecationWarning: The Sage wrappers around ppl are now superseded by
\n→the standalone pplpy.
Please use import 'ppl' instead of 'sage.libs.ppl'.
See http://trac.sagemath.org/23024 for details.
...
size 3 1 3 2
```

coefficient(v)
Return the coefficient of the variable v.

INPUT:

• v a Variable.

OUTPUT:
An integer.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: e = 3*x+1
sage: e.coefficient(x)
3
```

coefficients()
Return the coefficients of the linear expression.

OUTPUT:
A tuple of integers of length space_dimension().

EXAMPLES:
\texttt{sage: from sage.libs.ppl import Variable}
\texttt{sage: x = Variable(0); y = Variable(1)}
\texttt{sage: e = 3\times+5\times y+1}
\texttt{sage: e.coefficients()}
\texttt{(3, 5)}

\textbf{inhomogeneous_term()}
Return the inhomogeneous term of the linear expression.

\textbf{OUTPUT:}
Integer.

\textbf{EXAMPLES:}
\begin{verbatim}
\texttt{sage: from sage.libs.ppl import Variable, Linear_Expression}
\texttt{sage: Linear_Expression(10).inhomogeneous_term()}
10
\end{verbatim}

\textbf{is_zero()}
Test if \texttt{self} is the zero linear expression.

\textbf{OUTPUT:}
Boolean.

\textbf{EXAMPLES:}
\begin{verbatim}
\texttt{sage: from sage.libs.ppl import Variable, Linear_Expression}
\texttt{sage: Linear_Expression(0).is_zero()}
True
\texttt{sage: Linear_Expression(10).is_zero()}
False
\end{verbatim}

\textbf{space_dimension()}
Return the dimension of the vector space necessary for the linear expression.

\textbf{OUTPUT:}
Integer.

\textbf{EXAMPLES:}
\begin{verbatim}
\texttt{sage: from sage.libs.ppl import Variable}
\texttt{sage: x = Variable(0)}
\texttt{sage: y = Variable(1)}
\texttt{sage: ( x+y+1 ).space_dimension()}
2
\texttt{sage: ( x+y ).space_dimension()}
2
\texttt{sage: ( y+1 ).space_dimension()}
2
\texttt{sage: ( x +1 ).space_dimension()}
1
\texttt{sage: ( y+1-y ).space_dimension()}
2
\end{verbatim}

\textbf{class sage.libs.ppl.MIP_Problem}
Bases: sage.libs.ppl._mutable_or_immutable
wrapper for PPL's MIP_Problem class
An object of the class MIP_Problem represents a Mixed Integer (Linear) Program problem.

**INPUT:**

- dim – integer
- args – an array of the defining data of the MIP_Problem. For each element, any one of the following is accepted:
  - A `Constraint_System`
  - A `Linear_Expression`

**OUTPUT:**

A `MIP_Problem`.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0)
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2, cs, x + y)
sage: m.optimal_value()
10/3
sage: m.optimizing_point()
point(10/3, 0/3)
```

**OK()**

Check if all the invariants are satisfied.

**OUTPUT:**

True if and only if `self` satisfies all the invariants.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: m = MIP_Problem()
sage: m.add_space_dimensions_and_embed(2)
sage: m.add_constraint(x >= 0)
sage: m.OK()
True
```

**add_constraint(c)**

Adds a copy of constraint `c` to the MIP problem.

**EXAMPLES:**

(continues on next page)
add_constraints\( (cs)\)

Adds a copy of the constraints in \(cs\) to the MIP problem.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0)
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2)
sage: m.set_objective_function(x + y)
sage: m.add_constraints(cs)
sage: m.optimal_value()
10/3
```

add_space_dimensions_and_embed\( (m)\)

Adds \(m\) new space dimensions and embeds the old MIP problem in the new vector space.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0)
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2)
sage: m.set_objective_function(x + y)
sage: m.add_constraints(cs)
sage: m.optimal_value()
10/3
```

add_to_integer_space_dimensions\( (i\_vars)\)

Sets the variables whose indexes are in set \(i\_vars\) to be integer space dimensions.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Variables_Set, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0)
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2, cs, x + y)
sage: m.add_to_integer_space_dimensions(i_vars)
sage: m.space_dimension()
7
```
clear()

Reset the MIP_Problem to be equal to the trivial MIP_Problem.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0)
sage: cs.insert( y >= 0)
sage: cs.insert( 3 * x + 5 * y <= 10)
sage: m = MIP_Problem(2, cs, x + y)
sage: m.objective_function()
x0+x1
sage: m.clear()
sage: m.objective_function()
0
```

evaluate_objective_function(evaluating_point)

Return the result of evaluating the objective function on evaluating_point. ValueError thrown if self and evaluating_point are dimension-incompatible or if the generator evaluating_point is not a point.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem, Generator
sage: x = Variable(0)
sage: y = Variable(1)
sage: m = MIP_Problem()
sage: m.add_space_dimensions_and_embed(2)
sage: m.add_constraint(x >= 0)
sage: m.add_constraint(y >= 0)
sage: m.add_constraint(3 * x + 5 * y <= 10)
sage: m.set_objective_function(x + y)
sage: g = Generator.point(5 * x - 2 * y, 7)
sage: m.evaluate_objective_function(g)
3/7
sage: z = Variable(2)
sage: g = Generator.point(5 * x - 2 * z, 7)
sage: m.evaluate_objective_function(g)
Traceback (most recent call last):
...:
ValueError: PPL::MIP_Problem::evaluate_objective_function(p, n, d):
this and p are dimension incompatible.
```

is_satisfiable()

Check if the MIP_Problem is satisfiable

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
```

(continues on next page)
sage: y = Variable(1)
sage: m = MIP_Problem()
sage: m.add_space_dimensions_and_embed(2)
sage: m.add_constraint(x >= 0)
sage: m.add_constraint(y >= 0)
sage: m.add_constraint(3 * x + 5 * y <= 10)
sage: m.is_satisfiable()
True

**objective_function()**

Return the optimal value of the MIP_Problem.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0 )
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2, cs, x + y)
sage: m.objective_function()
x0+x1
```

**optimal_value()**

Return the optimal value of the MIP_Problem. ValueError thrown if self does not have an optimizing point, i.e., if the MIP problem is unbounded or not satisfiable.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0 )
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2, cs, x + y)
sage: m.optimal_value()
10/3
sage: cs = Constraint_System()
sage: cs.insert( x >= 0 )
sage: m = MIP_Problem(1, cs, x + x )
sage: m.optimal_value()
Traceback (most recent call last):
... ValueError: PPL::MIP_Problem::optimizing_point(): *this does not have an optimizing point.
```

**optimization_mode()**

Return the optimization mode used in the MIP_Problem.

It will return “maximization” if the MIP_Problem was set to MAXIMIZATION mode, and “minimization” otherwise.

**EXAMPLES:**
sage: from sage.libs.ppl import MIP_Problem
sage: m = MIP_Problem()
sage: m.optimization_mode()
'maximization'

**optimizing_point()**

Returns an optimal point for the MIP_Problem, if it exists.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: m = MIP_Problem()
sage: m.add_space_dimensions_and_embed(2)
sage: m.add_constraint(x >= 0)
sage: m.add_constraint(y >= 0)
sage: m.add_constraint(3 * x + 5 * y <= 10)
sage: m.set_objective_function(x + y)
sage: m.optimizing_point()
point(10/3, 0/3)
```

**set_objective_function**(obj)

Sets the objective function to obj.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: m = MIP_Problem()
sage: m.add_space_dimensions_and_embed(2)
sage: m.add_constraint(x >= 0)
sage: m.add_constraint(y >= 0)
sage: m.add_constraint(3 * x + 5 * y <= 10)
sage: m.set_objective_function(x + y)
sage: m.set_objective_function(10)
sage: m.optimal_value()
10/3
```

**set_optimization_mode**(mode)

Sets the optimization mode to mode.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import MIP_Problem
sage: m = MIP_Problem()
sage: m.optimization_mode()
'maximization'
sage: m.set_optimization_mode('minimization')
sage: m.optimization_mode()
'minimization'
```

**solve()**

Optimizes the MIP_Problem

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: m = MIP_Problem()
sage: m.add_space_dimensions_and_embed(2)
sage: m.add_constraint(x >= 0)
sage: m.add_constraint(y >= 0)
sage: m.add_constraint(3 * x + 5 * y <= 10)
sage: m.set_objective_function(x + y)
sage: m.solve()
```

14.1. Cython wrapper for the Parma Polyhedra Library (PPL) 155
```python
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: m = MIP_Problem()
sage: m.add_space_dimensions_and_embed(2)
sage: m.add_constraint(x >= 0)
sage: m.add_constraint(y >= 0)
sage: m.add_constraint(3 * x + 5 * y <= 10)
sage: m.set_objective_function(x + y)
sage: m.solve()
{'status': 'optimized'}
```

**space_dimension()**

Return the space dimension of the MIP_Problem.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, Constraint_System, MIP_Problem
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x >= 0)
sage: cs.insert( y >= 0 )
sage: cs.insert( 3 * x + 5 * y <= 10 )
sage: m = MIP_Problem(2, cs, x + y)
sage: m.space_dimension()
2
```

**class sage.libs.ppl.NNC_Polyhedron**

Bases: `sage.libs.ppl.Polyhedron`

Wrapper for PPL's NNC_Polyhedron class.

An object of the class NNC_Polyhedron represents a not necessarily closed (NNC) convex polyhedron in the vector space.

Note: Since NNC polyhedra are a generalization of closed polyhedra, any object of the class `C_Polyhedron` can be (explicitly) converted into an object of the class `NNC_Polyhedron`. The reason for defining two different classes is that objects of the class `C_Polyhedron` are characterized by a more efficient implementation, requiring less time and memory resources.

**INPUT:**

- `arg` – the defining data of the polyhedron. Any one of the following is accepted:
  - An non-negative integer. Depending on `degenerate_element`, either the space-filling or the empty polytope in the given dimension `arg` is constructed.
  - A `Constraint_System`.
  - A `Generator_System`.
  - A single `Constraint`.
  - A single `Generator`.
  - A `NNC_Polyhedron`.
  - A `C_Polyhedron`.
- `degenerate_element` – string, either 'universe' or 'empty'. Only used if `arg` is an integer.
A **C_Polyhedron**.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Constraint, Constraint_System, Generator,
    Generator_System, Variable, NNC_Polyhedron, point, ray, closure_point
sage: x = Variable(0)
sage: y = Variable(1)
sage: NNC_Polyhedron( 5*x-2*y > x+y-1 )  # A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point, 1 closure_point, 1 ray, 1 line
sage: cs = Constraint_System()
sage: cs.insert( x > 0 )
sage: cs.insert( y > 0 )
sage: NNC_Polyhedron(cs)  # A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point, 1 closure_point, 2 rays
sage: NNC_Polyhedron( point(x+y) )  # A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point
sage: gs = Generator_System()
sage: gs.insert( point(-y) )
sage: gs.insert( closure_point(-x-y) )
sage: gs.insert( ray(x) )
sage: p = NNC_Polyhedron(gs); p  # A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point, 1 closure_point, 1 ray
sage: p.minimized_constraints()  # Constraint_System {x1+1==0, x0+1>0}
```

Note that, by convention, every polyhedron must contain a point:

```python
sage: NNC_Polyhedron( closure_point(x+y) )
Traceback (most recent call last):
... ValueError: PPL::NNC_Polyhedron::NNC_Polyhedron(gs):
  *this is an empty polyhedron and the non-empty generator system gs contains no points.
```

**class** `sage.libs.ppl.Poly_Con_Relation`

*Wrapper for PPL’s Poly_Con_Relation class.*

**INPUT/OUTPUT:**

You must not construct *Poly_Con_Relation* objects manually. You will usually get them from *relation_with()*. You can also get pre-defined relations from the class methods *nothing()*, *is_disjoint()*, *strictly_intersects()*, *is_included()*, and *saturates()*.

**EXAMPLES:**

```python
sage: saturates = Poly_Con_Relation.saturates(); saturates
sage: is_included = Poly_Con_Relation.is_included(); is_included
sage: is_included.implies(saturates)
False
```

(continues on next page)
\texttt{sage:} saturates.implies(is\_included)  
\texttt{False}  
\texttt{sage:} rels = [  
\texttt{sage:} rels.append( Poly\_Con\_Relation.nothing() )  
\texttt{sage:} rels.append( Poly\_Con\_Relation.is\_disjoint() )  
\texttt{sage:} rels.append( Poly\_Con\_Relation.strictly\_intersects() )  
\texttt{sage:} rels.append( Poly\_Con\_Relation.is\_included() )  
\texttt{sage:} rels.append( Poly\_Con\_Relation.saturates() )  
\texttt{sage:} rels  
[\texttt{nothing, is\_disjoint, strictly\_intersects, is\_included, saturates}]  
\texttt{sage:} from sage.matrix.constructor import matrix  
\texttt{sage:} m = matrix(5,5)  
\texttt{sage:} for i, rel\_i in enumerate(rels):  
\texttt{....:} for j, rel\_j in enumerate(rels):  
\texttt{....:} m[i,j] = rel\_i.implies(rel\_j)  
\texttt{sage:} m  
\begin{bmatrix}  
1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 \\
\end{bmatrix}  
\texttt{OK} (\texttt{check\_non\_empty=}\texttt{False})  
Check if all the invariants are satisfied.  
\textbf{EXAMPLES:}  
\texttt{sage:} from sage.libs.ppl import Poly\_Con\_Relation  
\texttt{sage:} Poly\_Con\_Relation.nothing().OK()  
\texttt{True}  
\texttt{ascii\_dump()}  
Write an ASCII dump to stderr.  
\textbf{EXAMPLES:}  
\texttt{sage:} sage\_cmd = \texttt{'from sage.libs.ppl import Poly\_Con\_Relation\n'}  
\texttt{sage:} sage\_cmd += \texttt{'Poly\_Con\_Relation.nothing().ascii\_dump()\n'}  
\texttt{sage:} from sage.tests.cmdline import test\_executable  
\texttt{sage:} (out, err, ret) = test\_executable([\texttt{'sage', '-c', sage\_cmd}],\texttt{timeout=100})  
\texttt{# long time, indirect doctest}  
\texttt{sage:} print(err)  
\texttt{/... DeprecationWarning: The Sage wrappers around ppl are now superseded by the standalone pplpy.}  
\texttt{Please use import \texttt{\textquote{ppl}}}instead of \texttt{\textquote{sage.libs.ppl}}.\texttt{\textquote{.}}See \texttt{http://trac.sagemath.org/23024 for details.}  
\texttt{...}  
\texttt{NOTHING}  
\texttt{implies(y)}  
Test whether self implies y.  
\textbf{INPUT:}  
\begin{itemize}  
\item \texttt{y --} a \texttt{Poly\_Con\_Relation}.  
\end{itemize}  
\textbf{OUTPUT:}  
Boolean. True if and only if self implies y.
EXAMPLES:

```python
sage: from sage.libs.ppl import Poly_Con_Relation
sage: nothing = Poly_Con_Relation.nothing()
sage: nothing.implies( nothing )
True
```

**is_disjoint()**

Return the assertion “The polyhedron and the set of points satisfying the constraint are disjoint”.

**OUTPUT:**

A `Poly_Con_Relation`.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.is_disjoint()
```

**is_included()**

Return the assertion “The polyhedron is included in the set of points satisfying the constraint”.

**OUTPUT:**

A `Poly_Con_Relation`.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.is_included()
```

**nothing()**

Return the assertion that says nothing.

**OUTPUT:**

A `Poly_Con_Relation`.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.nothing()
```

**saturates()**

Return the assertion “”.

**OUTPUT:**

A `Poly_Con_Relation`.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.saturates()
```

**strictly_intersects()**

Return the assertion “The polyhedron intersects the set of points satisfying the constraint, but it is not included in it”.

14.1. Cython wrapper for the Parma Polyhedra Library (PPL)
A `Poly_Con_Relation`.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.strictly_intersects()
```

**class** `sage.libs.ppl.Poly_Gen_Relation`

**Bases:** `object`

Wrapper for PPL's `Poly_Con_Relation` class.

**INPUT/OUTPUT:**

You must not construct `Poly_Gen_Relation` objects manually. You will usually get them from `relation_with()`. You can also get pre-defined relations from the class methods `nothing()` and `subsumes()`.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Poly_Gen_Relation
sage: nothing = Poly_Gen_Relation.nothing(); nothing
nothing
sage: subsumes = Poly_Gen_Relation.subsumes(); subsumes
subsumes
sage: nothing.implies( subsumes )
False
sage: subsumes.implies( nothing )
True
```

**OK** *(check_non_empty=False)*

Check if all the invariants are satisfied.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Poly_Gen_Relation
sage: Poly_Gen_Relation.nothing().OK()
```

**ascii_dump()**

Write an ASCII dump to stderr.

**EXAMPLES:**

```python
sage: sage_cmd = 'from sage.libs.ppl import Poly_Gen_Relation
sage: sage_cmd += 'Poly_Gen_Relation.nothing().ascii_dump()\n'
sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd],
       →timeout=100) # long time, indirect doctest
sage: print(err)
// DeprecationWarning: The Sage wrappers around ppl are now superseded by
→the standalone pplpy.
Please use import 'ppl' instead of 'sage.libs.ppl'.
See http://trac.sagemath.org/23024 for details.
...
NOTHING
```
implies(y)
Test whether self implies y.

INPUT:
• y – a Poly_Gen_Relation.

OUTPUT:
Boolean. True if and only if self implies y.

EXAMPLES:

```python
sage: from sage.libs.ppl import Poly_Gen_Relation
sage: nothing = Poly_Gen_Relation.nothing()
```

nothing()
Return the assertion that says nothing.

OUTPUT:
A Poly_Gen_Relation.

EXAMPLES:

```python
sage: from sage.libs.ppl import Poly_Gen_Relation
```

subsumes()
Return the assertion “Adding the generator would not change the polyhedron”.

OUTPUT:
A Poly_Gen_Relation.

EXAMPLES:

```python
sage: from sage.libs.ppl import Poly_Gen_Relation
```

class sage.libs.ppl.Polyhedron
Bases: sage.libs.ppl._mutable_or_immutable

Wrapper for PPL’s Polyhedron class.

An object of the class Polyhedron represents a convex polyhedron in the vector space.

A polyhedron can be specified as either a finite system of constraints or a finite system of generators (see Section Representations of Convex Polyhedra) and it is always possible to obtain either representation. That is, if we know the system of constraints, we can obtain from this the system of generators that define the same polyhedron and vice versa. These systems can contain redundant members: in this case we say that they are not in the minimal form.

INPUT/OUTPUT:

This is an abstract base for C_Polyhedron and NNC_Polyhedron. You cannot instantiate this class.

OK (check_non_empty=False)
Check if all the invariants are satisfied.
The check is performed so as to intrude as little as possible. If the library has been compiled with run-time assertions enabled, error messages are written on std::cerr in case invariants are violated. This is useful for the purpose of debugging the library.

**INPUT:**

- `check_not_empty` – boolean. True if and only if, in addition to checking the invariants, `self` must be checked to be not empty.

**OUTPUT:**

True if and only if `self` satisfies all the invariants and either `check_not_empty` is False or `self` is not empty.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Linear_Expression, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: e = 3*x+2*y+1
sage: e.OK()
True
```

**add_constraint** *(c)*

Add a constraint to the polyhedron.

Adds a copy of constraint `c` to the system of constraints of `self`, without minimizing the result.

See also `add_constraints()`.

**INPUT:**

- `c` – the `Constraint` that will be added to the system of constraints of `self`.

**OUTPUT:**

This method modifies the polyhedron `self` and does not return anything.

Raises a `ValueError` if `self` and the constraint `c` are topology-incompatible or dimension-incompatible.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, C_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( y>=0 )
sage: p.add_constraint( x>=0 )
We just added a 1-d constraint to a 2-d polyhedron, this is fine. The other way is not::

sage: p = C_Polyhedron( x>=0 )
sage: p.add_constraint( y>=0 )
Traceback (most recent call last):
  ...
ValueError: PPL::C_Polyhedron::add_constraint(c):
this->space_dimension() == 1, c.space_dimension() == 2.
The constraint must also be topology-compatible, that is, :class:`C_Polyhedron` only allows non-strict inequalities::
```
```
sage: p = C_Polyhedron( x>=0 )
sage: p.add_constraint( x< 1 )
Traceback (most recent call last):
...  
ValueError: PPL::C_Polyhedron::add_constraint(c):
c is a strict inequality.

add_constraints(cs)
Add constraints to the polyhedron.

Adds a copy of constraints in cs to the system of constraints of self, without minimizing the result.

See also add_constraint().

INPUT:
- cs – the Constraint_System that will be added to the system of constraints of self.

OUTPUT:
This method modifies the polyhedron self and does not return anything.

Raises a ValueError if self and the constraints in cs are topology-incompatible or dimension-incompatible.

EXAMPLES:

sage: from sage.libs.ppl import Variable, C_Polyhedron, Constraint_System
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert(x>=0)
sage: cs.insert(y>=0)
sage: p = C_Polyhedron( y<=1 )
sage: p.add_constraints(cs)

We just added a 1-d constraint to a 2-d polyhedron, this is fine. The other way is not::

sage: p = C_Polyhedron( x<=1 )
sage: p.add_constraints(cs)
Traceback (most recent call last):
...  
ValueError: PPL::C_Polyhedron::add_recycled_constraints(cs):
this->space_dimension() == 1, cs.space_dimension() == 2.

The constraints must also be topology-compatible, that is, :class:`C_Polyhedron` only allows non-strict inequalities::

sage: p = C_Polyhedron( x>=0 )
sage: p.add_constraints( Constraint_System(x<0) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::add_recycled_constraints(cs):
cs contains strict inequalities.

add_generator(g)
Add a generator to the polyhedron.

Adds a copy of constraint c to the system of generators of self, without minimizing the result.
INPUT:

- g – the Generator that will be added to the system of Generators of self.

OUTPUT:

This method modifies the polyhedron self and does not return anything.

Raises a ValueError if self and the generator g are topology-incompatible or dimension-incompatible, or if self is an empty polyhedron and g is not a point.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, C_Polyhedron, point, closure_point
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron(1, 'empty')
sage: p.add_generator( point(0*x) )

We just added a 1-d generator to a 2-d polyhedron, this is fine. The other way is not:

```python
def

```python
sage: p = C_Polyhedron(1, 'empty')
sage: p.add_generator( point(0*y) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::add_generator(g):
  this->space_dimension() == 1, g.space_dimension() == 2.

The constraint must also be topology-compatible, that is, :class:`C_Polyhedron` does not allow :func:`closure_point` generators:

```python
sage: p = C_Polyhedron( point(0*x+0*y) )
sage: p.add_generator( closure_point(0*x) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::add_generator(g):
  g is a closure point.
```

Finally, ever non-empty polyhedron must have at least one point generator:

```python
sage: p = C_Polyhedron(3, 'empty')
sage: p.add_generator( ray(x) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::add_generator(g):
  *this is an empty polyhedron and g is not a point.
```

**add_generators(gs)**

Add generators to the polyhedron.

Adds a copy of the generators in gs to the system of generators of self, without minimizing the result.

See also **add_generator()**.

INPUT:

- gs – the Generator_System that will be added to the system of constraints of self.

OUTPUT:
This method modifies the polyhedron `self` and does not return anything.

Raising a `ValueError` if `self` and one of the generators in `gs` are topology-incompatible or dimension-incompatible, or if `self` is an empty polyhedron and `gs` does not contain a point.

**EXAMPLES:**
```python
sage: from sage.libs.ppl import Variable, C_Polyhedron, Generator_System,
                point, ray, closure_point
sage: x = Variable(0)
sage: y = Variable(1)
sage: gs = Generator_System()
sage: gs.insert(point(0*x+0*y))
sage: gs.insert(point(1*x+1*y))
sage: p = C_Polyhedron(2, 'empty')
sage: p.add_generators(gs)
We just added a 1-d constraint to a 2-d polyhedron, this is fine. The other way is not:
```
sage: p = C_Polyhedron(1, 'empty')
sage: p.add_generators(gs)
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::add_recycled_generators(gs):
this->space_dimension() == 1, gs.space_dimension() == 2.
The constraints must also be topology-compatible, that is, `C_Polyhedron` does not allow `closure_point` generators:
```sage: p = C_Polyhedron(point(0*x+0*y))
sage: p.add_generators( Generator_System(closure_point(x)) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::add_recycled_generators(gs):
gs contains closure points.
```

**add_space_dimensions_and_embed** *(m)*
Add `m` new space dimensions and embed `self` in the new vector space.

The new space dimensions will be those having the highest indexes in the new polyhedron, which is characterized by a system of constraints in which the variables running through the new dimensions are not constrained. For instance, when starting from the polyhedron `P` and adding a third space dimension, the result will be the polyhedron

\[
\left\{ (x, y, z)^T \in \mathbb{R}^3 \mid (x, y)^T \in P \right\}
\]

**INPUT:**
- `m` – integer.

**OUTPUT:**
This method assigns the embedded polyhedron to `self` and does not return anything.

Raising a `ValueError` if adding `m` new space dimensions would cause the vector space to exceed dimension `self.max_space_dimension()`.

**EXAMPLES:**
sage: from sage.libs.ppl import Variable, C_Polyhedron, point
sage: x = Variable(0)

sage: p = C_Polyhedron( point(3*x) )

sage: p.add_space_dimensions_and_embed(1)

sage: p.minimized_generators()
Generator_System {line(0, 1), point(3/1, 0/1)}

sage: p.add_space_dimensions_and_embed( p.max_space_dimension() )
Traceback (most recent call last):
... ValueError: PPL::C_Polyhedron::add_space_dimensions_and_embed(m):
    adding m new space dimensions exceeds the maximum allowed space dimension.

add_space_dimensions_and_project(m)

Add m new space dimensions and embed self in the new vector space.

The new space dimensions will be those having the highest indexes in the new polyhedron, which is
characterized by a system of constraints in which the variables running through the new dimensions are all
constrained to be equal to 0. For instance, when starting from the polyhedron $P$ and adding a third space
dimension, the result will be the polyhedron

$$\left\{(x,y,0)^T \in \mathbb{R}^3 \mid (x,y)^T \in P\right\}$$

INPUT:

• m – integer.

OUTPUT:

This method assigns the projected polyhedron to self and does not return anything.

Raises a ValueError if adding m new space dimensions would cause the vector space to exceed dimen-
sion self.max_space_dimension().

EXAMPLES:

affine_dimension()  

Return the affine dimension of self.

OUTPUT:

An integer. Returns 0 if self is empty. Otherwise, returns the affine dimension of self.

EXAMPLES:

...
sage: p.affine_dimension()
1

ascii_dump()
Write an ASCII dump to stderr.

EXAMPLES:

sage: sage_cmd = 'from sage.libs.ppl import C_Polyhedron, Variable
            x = Variable(0)
            y = Variable(1)
            p = C_Polyhedron(3*x+2*y==1)
            p.minimized_generators()
            p.ascii_dump()

sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd],
                                    timeout=100) # long time, indirect doctest
sage: print(err) # long time
/... DeprecationWarning: The Sage wrappers around ppl are now superseded by
   the standalone pplpy. Please use import 'ppl' instead of 'sage.libs.ppl'.
   See http://trac.sagemath.org/23024 for details.
   ...
   space_dim 2
   -ZE -EM +CM +GM +CS +GS -CP -GP -SC +SG
   con_sys (up-to-date)
   topology NECESSARILY_CLOSED
   2 x 2 SPARSE (sorted)
   index_first_pending 2
   size 3 -1 3 2 = (C)
   size 3 1 0 0 >= (C)
   gen_sys (up-to-date)
   topology NECESSARILY_CLOSED
   2 x 2 DENSE (not_sorted)
   index_first_pending 2
   size 3 0 2 -3 L (C)
   size 3 2 0 1 P (C)
   sat_c
   0 x 0
   sat_g
   2 x 2
   0 0
   0 1

bounds_from_above(expr)
Test whether the expr is bounded from above.

INPUT:

• expr – a LinearExpression

OUTPUT:

Boolean. Returns True if and only if expr is bounded from above in self.

Raises a ValueError if expr and this are dimension-incompatible.
EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, C_Polyhedron, Linear_Expression
sage: x = Variable(0); y = Variable(1)
```

```python
sage: p = C_Polyhedron(y<=0)
```

```python
sage: p.bounds_from_above(x+1)
False
```

```python
sage: p.bounds_from_above(Linear_Expression(y))
```

```python
True
```

```python
sage: p = C_Polyhedron(x<=0)
```

```python
sage: p.bounds_from_below(y+1)
```

```python
Traceback (most recent call last):
  ... ValueError: PPL::C_Polyhedron::bounds_from_above(e):
this->space_dimension() == 1, e.space_dimension() == 2.
```

**bounds_from_below** (*expr*)

Test whether the *expr* is bounded from above.

**INPUT:**

* expr – a `Linear_Expression`

**OUTPUT:**

Boolean. Returns True if and only if *expr* is bounded from above in *self*. Raises a `ValueError` if *expr* and *this* are dimension-incompatible.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, C_Polyhedron, Linear_Expression
sage: x = Variable(0); y = Variable(1)
```

```python
sage: p = C_Polyhedron(y>=0)
```

```python
sage: p.bounds_from_below(x+1)
False
```

```python
sage: p.bounds_from_below(Linear_Expression(y))
```

```python
True
```

```python
sage: p = C_Polyhedron(x<=0)
```

```python
sage: p.bounds_from_below(y+1)
```

```python
Traceback (most recent call last):
  ... ValueError: PPL::C_Polyhedron::bounds_from_below(e):
this->space_dimension() == 1, e.space_dimension() == 2.
```

**concatenate_assign** (*y*)

Assign to *self* the concatenation of *self* and *y*.

This function returns the Cartesian product of *self* and *y*.

Viewing a polyhedron as a set of tuples (its points), it is sometimes useful to consider the set of tuples obtained by concatenating an ordered pair of polyhedra. Formally, the concatenation of the polyhedra $P$ and $Q$ (taken in this order) is the polyhedron such that

\[ R = \{(x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1})^T \in \mathbb{R}^{n+m} \mid (x_0, \ldots, x_{n-1})^T \in P, (y_0, \ldots, y_{m-1})^T \in Q\} \]

Another way of seeing it is as follows: first embed polyhedron $P$ into a vector space of dimension $n + m$ and then add a suitably renamed-apart version of the constraints defining $Q$.

**INPUT:**

* m – integer.
OUTPUT:

This method assigns the concatenated polyhedron to `self` and does not return anything.

Raises a `ValueError` if `self` and `y` are topology-incompatible or if adding `y`.

space_dimension() new space dimensions would cause the vector space to exceed dimension`self.max_space_dimension()`.

EXAMPLES:

```sage
def from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron, point
x = Variable(0)
p1 = C_Polyhedron( point(1*x) )
p2 = C_Polyhedron( point(2*x) )
p1.concatenate_assign(p2)
```

The polyhedra must be topology-compatible and not exceed the maximum space dimension:

```sage
def p1.concatenate_assign( NNC_Polyhedron(1, 'universe') )
```

```sage
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::concatenate_assign(y):
y is a NNC_Polyhedron.
```

```sage
def p1.concatenate_assign( C_Polyhedron(p1.max_space_dimension(), 'empty') )
```

```sage
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::concatenate_assign(y):
concatenation exceeds the maximum allowed space dimension.
```

constrains(var)

Test whether `var` is constrained in `self`.

INPUT:

- `var` — a `Variable`.

OUTPUT:

Boolean. Returns `True` if and only if `var` is constrained in `self`.

Raises a `ValueError` if `var` is not a space dimension of `self`.

EXAMPLES:

```sage
def from sage.libs.ppl import Variable, C_Polyhedron
x = Variable(0)
p = C_Polyhedron(1, 'universe')
p.constrains(x)
```

```sage
False
```

```sage
def p.constrains(x)
```

```sage
False
```

```sage
def p.constrains(x)
```

```sage
True
```

```sage
def y = Variable(1)
```

```sage
def p.constrains(y)
```

```sage
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::constrains(v):
this->space_dimension() == 1, v.space_dimension() == 2.
```
constraints()  
Returns the system of constraints.  
See also minimized_constraints().  
OUTPUT:  
A Constraint_System.  
EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( y>=0 )
sage: p.add_constraint( x>=0 )
sage: p.add_constraint( x+y>=0 )
sage: p.constraints()
Constraint_System {x1>=0, x0>=0, x0+x1>=0}
sage: p.minimized_constraints()
Constraint_System {x1>=0, x0>=0}
```

contains(y)  
Test whether self contains y.  
INPUT:  
• y – a Polyhedron.  
OUTPUT:  
Boolean. Returns True if and only if self contains y.  
Raises a ValueError if self and y are topology-incompatible or dimension-incompatible.  
EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p0 = C_Polyhedron( x>=0 )
sage: p1 = C_Polyhedron( x>=1 )
sage: p0.contains(p1)
True
sage: p1.contains(p0)
False
```

Errors are raised if the dimension or topology is not compatible:

```
sage: p0.contains(C_Polyhedron(y>=0))
Traceback (most recent call last):
...  
ValueError: PPL::C_Polyhedron::contains(y):
this->space_dimension() == 1, y.space_dimension() == 2.
sage: p0.contains(NNC_Polyhedron(x>0))
Traceback (most recent call last):
...  
ValueError: PPL::C_Polyhedron::contains(y):
y is a NNC_Polyhedron.
```

contains_integer_point()  
Test whether self contains an integer point.
OUTPUT:

Boolean. Returns True if and only if self contains an integer point.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, NNC_Polyhedron
sage: x = Variable(0)
sage: p = NNC_Polyhedron(x>0)
sage: p.add_constraint(x<1)
sage: p.contains_integer_point()
False
sage: p.topological_closure_assign()
```

```python
sage: p.contains_integer_point()
True
```

**difference_assign**(y)
Assign to self the poly-difference of self and y.

For any pair of NNC polyhedra $P_1$ and $P_2$ the convex polyhedral difference (or poly-difference) of $P_1$ and $P_2$ is defined as the smallest convex polyhedron containing the set-theoretic difference $P_1 \setminus P_2$ of $P_1$ and $P_2$.

In general, even if $P_1$ and $P_2$ are topologically closed polyhedra, their poly-difference may be a convex polyhedron that is not topologically closed. For this reason, when computing the poly-difference of two C_Polyhedron, the library will enforce the topological closure of the result.

**INPUT:**

* y — a Polyhedron

**OUTPUT:**

This method assigns the poly-difference to self and does not return anything.

 Raises a ValueError if self and y are topology-incompatible or dimension-incompatible.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, C_Polyhedron, point, closure_point,
˓→NNC_Polyhedron
sage: x = Variable(0)
sage: p = NNC_Polyhedron( point(0*x) )
sage: p.add_generator( point(1*x) )
sage: p.poly_difference_assign(NNC_Polyhedron( point(0*x) ))
sage: p.minimized_constraints()
Constraint_System {-x0+1>=0, x0>0}
```

The poly-difference of C_polyhedron is really its closure:

```python
sage: p = C_Polyhedron( point(0*x) )
sage: p.add_generator( point(1*x) )
sage: p.poly_difference_assign(C_Polyhedron( point(0*x) ))
sage: p.minimized_constraints()
Constraint_System {x0>=0, -x0+1>=0}
```

self and y must be dimension- and topology-compatible, or an exception is raised:

```python
sage: y = Variable(1)
sage: p.poly_difference_assign( C_Polyhedron(y>=0) )
Traceback (most recent call last):
```
drop_some_non_integer_points()
Possibly tighten self by dropping some points with non-integer coordinates.

The modified polyhedron satisfies:

• it is (not necessarily strictly) contained in the original polyhedron.
• integral vertices (generating points with integer coordinates) of the original polyhedron are not re-
  moved.

Note: The modified polyhedron is not necessarily a lattice polyhedron; Some vertices will, in general, still be rational. Lattice points interior to the polyhedron may be lost in the process.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, NNC_Polyhedron, Constraint_System
sage: x = Variable(0)
sage: y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x>=0 )
sage: cs.insert( y>=0 )
sage: cs.insert( 3*x+2*y<5 )
sage: p = NNC_Polyhedron(cs)
sage: p.minimized_generators()
Generator_System {point(0/1, 0/1), closure_point(0/2, 5/2), closure_point(5/3, 0/3)}
sage: p.drop_some_non_integer_points()
sage: p.minimized_generators()
Generator_System {point(0/1, 0/1), point(0/1, 2/1), point(4/3, 0/3)}
```

generators()
Returns the system of generators.

See also minimized_generators().

OUTPUT:

A Generator_System.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, C_Polyhedron, point
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron(3,'empty')
sage: p.add_generator( point(-x-y) )
sage: p.add_generator( point(0) )
sage: p.add_generator( point(+x+y) )
```

(continues on next page)
**intersection_assign***(y)**

Assign to *self* the intersection of *self* and *y*.

**INPUT:**

- *y* – a *Polyhedron*

**OUTPUT:**

This method assigns the intersection to *self* and does not return anything.

Raises a **ValueError** if *self* and *y* are topology-incompatible or dimension-incompatible.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( 1*x+0*y >= 0 )
sage: p.intersection_assign( C_Polyhedron(y>=0) )
sage: p.constraints()
Constraint_System {x0>=0, x1>=0}
sage: z = Variable(2)
sage: p.intersection_assign( C_Polyhedron(z>=0) )
Traceback (most recent call last):
 ... ValueError: PPL::C_Polyhedron::intersection_assign(y):
this->space_dimension() == 2, y.space_dimension() == 3.
```

**is_bounded()**

Test whether *self* is bounded.

**OUTPUT:**

Boolean. Returns **True** if and only if *self* is a bounded polyhedron.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, NNC_Polyhedron, point, closure_-
  →point, ray
sage: x = Variable(0)
sage: p = NNC_Polyhedron( point(0+x) )
sage: p.add_generator( closure_point(1*x) )
sage: p.is_bounded()
True
sage: p.add_generator( ray(1*x) )
sage: p.is_bounded()
False
```
**is_discrete()**

Test whether `self` is discrete.

**OUTPUT:**

Boolean. Returns `True` if and only if `self` is discrete.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, C_Polyhedron, point, ray
sage: x = Variable(0); y = Variable(1)
sage: p = C_Polyhedron( point(1*x+2*y) )
sage: p.is_discrete()  # True
sage: p.add_generator( point(x) )
sage: p.is_discrete()  # False
```

**is_disjoint_from(y)**

Tests whether `self` and `y` are disjoint.

**INPUT:**

- `y` — a `Polyhedron`.

**OUTPUT:**

Boolean. Returns `True` if and only if `self` and `y` are disjoint. `Ray` raises a `ValueError` if `self` and `y` are topology-incompatible or dimension-incompatible.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron
sage: x = Variable(0); y = Variable(1)
sage: C_Polyhedron(x<=0).is_disjoint_from( C_Polyhedron(x>=1) )  # True
sage: poly_1d = C_Polyhedron(x<=0)
sage: poly_2d = C_Polyhedron(x+0*y>=1)
sage: poly_1d.is_disjoint_from(poly_2d)
Traceback (most recent call last):
  ... ValueError: PPL::C_Polyhedron::intersection_assign(y):
    this->space_dimension() == 1, y.space_dimension() == 2.
```

This is not allowed:

```python
sage: x = Variable(0); y = Variable(1)
sage: poly_1d = C_Polyhedron(x<=0)
sage: poly_2d = C_Polyhedron(x+0*y>=1)
sage: poly_1d.is_disjoint_from(poly_2d)
Traceback (most recent call last):
  ... ValueError: PPL::C_Polyhedron::intersection_assign(y):
    this->space_dimension() == 1, y.space_dimension() == 2.
```

Nor is this:

```python
sage: x = Variable(0); y = Variable(1)
sage: c_poly = C_Polyhedron( x<=0 )
sage: nnc_poly = NNC_Polyhedron( x >0 )
sage: c_poly.is_disjoint_from(nnc_poly)
Traceback (most recent call last):
  ... ValueError: PPL::C_Polyhedron::intersection_assign(y):
    y is a NNC_Polyhedron.
sage: NNC_Polyhedron(c_poly).is_disjoint_from(nnc_poly)
True
```
is_empty()  
Test if self is an empty polyhedron.

OUTPUT:  
Boolean.

EXAMPLES:

```python
sage: from sage.libs.ppl import C_Polyhedron
sage: C_Polyhedron(3, 'empty').is_empty()  
True
sage: C_Polyhedron(3, 'universe').is_empty()  
False
```

is_topologically_closed()  
Tests if self is topologically closed.

OUTPUT:  
Retruns True if and only if self is a topologically closed subset of the ambient vector space.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron
sage: x = Variable(0); y = Variable(1)
```

```python
sage: C_Polyhedron(3, 'universe').is_topologically_closed()  
True
sage: C_Polyhedron( x>=1 ).is_topologically_closed()  
True
sage: NNC_Polyhedron( x>1 ).is_topologically_closed()  
False
```

is_universe()  
Test if self is a universe (space-filling) polyhedron.

OUTPUT:  
Boolean.

EXAMPLES:

```python
sage: from sage.libs.ppl import C_Polyhedron
sage: C_Polyhedron(3, 'empty').is_universe()  
False
sage: C_Polyhedron(3, 'universe').is_universe()  
True
```

max_space_dimension()  
Return the maximum space dimension all kinds of Polyhedron can handle.

OUTPUT:  
Integer.

EXAMPLES:

```python
sage: from sage.libs.ppl import C_Polyhedron
sage: C_Polyhedron(1, 'empty').max_space_dimension()  
# random output
1152921504606846974
```

(continues on next page)
maximize(expr)
Maximize expr.

INPUT:

• expr – a LinearExpression.

OUTPUT:

A dictionary with the following keyword:value pair:

• 'bounded': Boolean. Whether the linear expression expr is bounded from above on self.

If expr is bounded from above, the following additional keyword:value pairs are set to provide information about the supremum:

• 'sup_n': Integer. The numerator of the supremum value.

• 'sup_d': Non-zero integer. The denominator of the supremum value.

• 'maximum': Boolean. True if and only if the supremum is also the maximum value.

• 'generator': a Generator. A point or closure point where expr reaches its supremum value.

EXAMPLES:

sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron, Constraint_System, Linear_Expression
sage: x = Variable(0); y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x>=0 )
sage: cs.insert( y>=0 )
sage: cs.insert( 3*x+5*y<=10 )
sage: p = C_Polyhedron(cs)
sage: p.maximize( x+y )
{'bounded': True, 'generator': point(10/3, 0/3), 'maximum': True, 'sup_d': 3, 'sup_n': 10}

Unbounded case:

sage: cs = Constraint_System()
sage: cs.insert( x>0 )
sage: p = NNC_Polyhedron(cs)
sage: p.maximize( +x )
{'bounded': False}
sage: p.maximize( -x )
{'bounded': True, 'generator': closure_point(0/1), 'maximum': False, 'sup_d': 1, 'sup_n': 0}

minimize(expr)
Minimize expr.
INPUT:

- `expr` - a `Linear_Expression`.

OUTPUT:

A dictionary with the following keyword:value pair:

- `'bounded'` - Boolean. Whether the linear expression `expr` is bounded from below on `self`.

If `expr` is bounded from below, the following additional keyword:value pairs are set to provide information about the infimum:

- `'inf_n'` - Integer. The numerator of the infimum value.
- `'inf_d'` - Non-zero integer. The denominator of the infimum value.
- `'minimum'` - Boolean. `True` if and only if the infimum is also the minimum value.
- `'generator'` - a `Generator`. A point or closure point where `expr` reaches its infimum value.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron,
constraint_System, Linear_Expression
sage: x = Variable(0); y = Variable(1)
sage: cs = Constraint_System()
sage: cs.insert( x>=0 )
sage: cs.insert( y>=0 )
sage: cs.insert( 3*x+5*y<=10 )
sage: p = C_Polyhedron(cs)
sage: p.minimize( x+y )
{'bounded': True,
 'generator': point(0/1, 0/1),
 'inf_d': 1,
 'inf_n': 0,
 'minimum': True}
```

Unbounded case:

```python
sage: cs = Constraint_System()
sage: cs.insert( x>0 )
sage: p = NNC_Polyhedron(cs)
sage: p.minimize( +x )
{'bounded': True,
 'generator': closure_point(0/1),
 'inf_d': 1,
 'inf_n': 0,
 'minimum': False}
sage: p.minimize( -x )
{'bounded': False}
```

`minimized_constraints()`

Returns the minimized system of constraints.

See also `constraints()`.

OUTPUT:

A `Constraint_System`.

EXAMPLES:
sage: from sage.libs.ppl import Variable, C_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( y>=0 )
sage: p.add_constraint( x>=0 )
sage: p.add_constraint( x+y>=0 )

minimized_generators()
Returns the minimized system of generators.
See also generators().

OUTPUT:
A Generator_System.

EXAMPLES:

sage: from sage.libs.ppl import Variable, C_Polyhedron, point
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron(3,'empty')
sage: p.add_generator( point(-x-y) )
sage: p.add_generator( point(0) )
sage: p.add_generator( point(+x+y) )

poly_difference_assign (y)
Assign to self the poly-difference of self and y.

For any pair of NNC polyhedra $P_1$ and $P_2$ the convex polyhedral difference (or poly-difference) of $P_1$ and $P_2$ is defined as the smallest convex polyhedron containing the set-theoretic difference $P_1 \setminus P_2$ of $P_1$ and $P_2$.

In general, even if $P_1$ and $P_2$ are topologically closed polyhedra, their poly-difference may be a convex polyhedron that is not topologically closed. For this reason, when computing the poly-difference of two C_Polyhedron, the library will enforce the topological closure of the result.

INPUT:

• y – a Polyhedron

OUTPUT:

This method assigns the poly-difference to self and does not return anything.

 Raises a ValueError if self and y are topology-incompatible or dimension-incompatible.

EXAMPLES:

(continues on next page)
sage: p = NNC_Polyhedron( point(0*x) )
sage: p.add_generator( point(1*x) )
sage: p.poly_difference_assign( NNC_Polyhedron( point(0*x) ) )
sage: p.minimized_constraints()
Constraint_System {-x0+1>=0, x0>0}

The poly-difference of \texttt{C\_polyhedron} is really its closure:

sage: p = C_Polyhedron( point(0*x) )
sage: p.add_generator( point(1*x) )
sage: p.poly_difference_assign(C_Polyhedron( point(0*x) ) )
sage: p.minimized_constraints()
Constraint_System {x0>=0, -x0+1>=0}

\texttt{self} and \texttt{y} must be dimension- and topology-compatible, or an exception is raised:

sage: y = Variable(1)
sage: p.poly_difference_assign( C_Polyhedron(y>=0) )
Traceback (most recent call last):
  ...  
ValueError: PPL::C_Polyhedron::poly_difference_assign(y):
  this->space_dimension() == 1, y.space_dimension() == 2.
sage: p.poly_difference_assign( NNC_Polyhedron(x+y<1) )
Traceback (most recent call last):
  ...  
ValueError: PPL::C_Polyhedron::poly_difference_assign(y):
  y is a NNC_Polyhedron.

\texttt{poly\_hull\_assign}(y)
Assign to \texttt{self} the poly-hull of \texttt{self} and \texttt{y}.

For any pair of NNC polyhedra \( P_1 \) and \( P_2 \), the convex polyhedral hull (or poly-hull) of is the smallest NNC polyhedron that includes both \( P_1 \) and \( P_2 \). The poly-hull of any pair of closed polyhedra in is also closed.

INPUT:

* \texttt{y} – a \texttt{Polyhedron}

OUTPUT:

This method assigns the poly-hull to \texttt{self} and does not return anything.

Raises a \texttt{ValueError} if \texttt{self} and \texttt{y} are topology-incompatible or dimension-incompatible.

EXAMPLES:

sage: from sage.libs.ppl import Variable, C_Polyhedron, point, NNC_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( point(1*x+0*y) )
sage: p.poly_hull_assign( C_Polyhedron( point(0*x+1*y) ) )
sage: p.generators()
Generator_System {point(0/1, 1/1), point(1/1, 0/1)}

\texttt{self} and \texttt{y} must be dimension- and topology-compatible, or an exception is raised:

sage: z = Variable(2)
sage: p.poly_hull_assign( C_Polyhedron(z>=0) )
relation_with(arg)

Return the relations holding between the polyhedron self and the generator or constraint arg.

INPUT:

• arg — a Generator or a Constraint.

OUTPUT:

A Poly_Gen_Relation or a Poly_Con_Relation according to the type of the input.

Raises ValueError if self and the generator/constraint arg are dimension-incompatible.

EXAMPLES:

sage: from sage.libs.ppl import Variable, C_Polyhedron, point, ray, Poly_Con_Relation

sage: x = Variable(0); y = Variable(1)
sage: p = C_Polyhedron(2, 'empty')

sage: p.add_generator( point(x*1+y*0) )
sage: p.add_generator( point(0*x+1*y) )

sage: p.minimized_constraints()
Constraint_System {x0+x1-1==0, -x1+1>=0, x1>=0}

sage: p.relation_with( point(1*x+1*y) )
nothing

sage: p.relation_with( point(1*x+1*y, 2) )
subsumes

sage: p.relation_with( x+y==1 )
is_disjoint

sage: p.relation_with( x==y )
strictly_intersects

sage: p.relation_with( x+y<=1 )
is_included, saturates

sage: p.relation_with( x+y<1 )
is_disjoint, saturates

In a Sage program you will usually use relation_with() together with implies() or implies(), for example:

sage: p.relation_with( x+y<1 ).implies(Poly_Con_Relation.saturates())
True

You can only get relations with dimension-compatible generators or constraints:

sage: z = Variable(2)
sage: p.relation_with( point(x+y+z) )
Traceback (most recent call last):
...
remove_higher_space_dimensions\((new\_dimension)\)
Remove the higher dimensions of the vector space so that the resulting space will have dimension new\_dimension.

OUTPUT:
This method modifies self and does not return anything.

Raises a ValueError if new\_dimensions is greater than the space dimension of self.

EXAMPLES:

```sage
from sage.libs.ppl import C_Polyhedron, Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron(3*x+0*y==2)
sage: p.remove_higher_space_dimensions(1)
sage: p.minimized_constraints()
Constraint_System {3*x0-2==0}
sage: p.remove_higher_space_dimensions(2)
Traceback (most recent call last):
... ValueError: PPL::C_Polyhedron::remove_higher_space_dimensions(nd):
this->space_dimension() == 1, required space dimension == 2.
```

space\_dimension\()
Return the dimension of the vector space enclosing self.

OUTPUT:
Integer.

EXAMPLES:

```sage
from sage.libs.ppl import Variable, C_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( 5*x-2*y >= x+y-1 )
sage: p.space_dimension()
2
```

strictly\_contains\(y\)
Test whether self strictly contains y.

INPUT:
• y – a Polyhedron.

OUTPUT:
Boolean. Returns True if and only if self contains y and self does not equal y.

Raises a ValueError if self and y are topology-incompatible or dimension-incompatible.
EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p0 = C_Polyhedron( x>=0 )
sage: p1 = C_Polyhedron( x>=1 )
 sage: p0.strictly_contains(p1)
  True
 sage: p1.strictly_contains(p0)
  False
```

Errors are raised if the dimension or topology is not compatible:

```python
sage: p0.strictly_contains(C_Polyhedron(y>=0))
Traceback (most recent call last):
...  ValueError: PPL::C_Polyhedron::contains(y):
this->space_dimension() == 1, y.space_dimension() == 2.
sage: p0.strictly_contains(NNC_Polyhedron(x>0))
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::contains(y):
y is a NNC_Polyhedron.
```

topological_closure_assign()
Assign to self its topological closure.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, NNC_Polyhedron
sage: x = Variable(0)
sage: p = NNC_Polyhedron(x>0)
sage: p.is_topologically_closed()
  False
 sage: p.topological_closure_assign()
 sage: p.is_topologically_closed()
  True
 sage: p.minimized_constraints()
  Constraint_System {x0>=0}
```

unconstrain(var)
Compute the cylindrification of self with respect to space dimension var.

INPUT:

- var -- a `Variable`. The space dimension that will be unconstrained. Exceptions:

OUTPUT:

This method assigns the cylindrification to self and does not return anything.

Raises a `ValueError` if var is not a space dimension of self.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, C_Polyhedron, point
def main():
    x = Variable(0)
y = Variable(1)
p = C_Polyhedron( point(x+y) ); p
main()
```
A 0-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 point
\begin{verbatim}
sage: p.unconstrain(x); p
\end{verbatim}
A 1-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 point, 1 → line
\begin{verbatim}
sage: z = Variable(2)
sage: p.unconstrain(z)
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::unconstrain(var):
this->space_dimension() == 2, required space dimension == 3.
\end{verbatim}

**upper_bound_assign**(*y*)
Assign to *self* the poly-hull of *self* and *y*.

For any pair of NNC polyhedra $P_1$ and $P_2$, the convex polyhedral hull (or poly-hull) of is the smallest NNC polyhedron that includes both $P_1$ and $P_2$. The poly-hull of any pair of closed polyhedra in is also closed.

**INPUT:**

- *y* – a *Polyhedron*

**OUTPUT:**

This method assigns the poly-hull to *self* and does not return anything.

Raises a *ValueError* if *self* and *y* are topology-incompatible or dimension-incompatible.

**EXAMPLES:**

\begin{verbatim}
sage: from sage.libs.ppl import Variable, C_Polyhedron, point, NNC_Polyhedron
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( point(1*x+0*y) )
sage: p.poly_hull_assign( C_Polyhedron( point(0*x+1*y) ) )
sage: p.generators()
Generator_System {point(0/1, 1/1), point(1/1, 0/1)}
\end{verbatim}

*self* and *y* must be dimension- and topology-compatible, or an exception is raised:

\begin{verbatim}
sage: z = Variable(2)
sage: p.poly_hull_assign( C_Polyhedron(z>=0) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::poly_hull_assign(y):
this->space_dimension() == 2, y.space_dimension() == 3.
sage: p.poly_hull_assign( NNC_Polyhedron(x+y<1) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::poly_hull_assign(y):
y is a NNC_Polyhedron.
\end{verbatim}

**class** sage.libs.ppl.Variable

**Bases:** object

Wrapper for PPL’s *Variable* class.

A dimension of the vector space.
An object of the class `Variable` represents a dimension of the space, that is one of the Cartesian axes. Variables are used as basic blocks in order to build more complex linear expressions. Each variable is identified by a non-negative integer, representing the index of the corresponding Cartesian axis (the first axis has index 0). The space dimension of a variable is the dimension of the vector space made by all the Cartesian axes having an index less than or equal to that of the considered variable; thus, if a variable has index \( i \), its space dimension is \( i + 1 \).

**INPUT:**

- \( i \) – integer. The index of the axis.

**OUTPUT:**

A `Variable`.

**EXAMPLES:**

```sage
from sage.libs.ppl import Variable
x = Variable(123)
x.id()  # 123
x     # x123
```

Note that the “meaning” of an object of the class `Variable` is completely specified by the integer index provided to its constructor: be careful not to be mislead by C++ language variable names. For instance, in the following example the linear expressions \( e_1 \) and \( e_2 \) are equivalent, since the two variables \( x \) and \( z \) denote the same Cartesian axis:

```sage
x = Variable(0)
y = Variable(1)
z = Variable(0)
e1 = x + y; e1
x0+x1
e2 = y + z; e2
x0+x1
e1 - e2
0
```

**OK()**

Checks if all the invariants are satisfied.

**OUTPUT:**

Boolean.

**EXAMPLES:**

```sage
from sage.libs.ppl import Variable
x = Variable(0)
x.OK()  # True
```

**id()**

Return the index of the Cartesian axis associated to the variable.

**EXAMPLES:**

```sage
from sage.libs.ppl import Variable
x = Variable(123)
```
space_dimension()
Return the dimension of the vector space enclosing self.

OUTPUT:
Integer. The returned value is self.id()+1.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: x.space_dimension()
1
```

class sage.libs.ppl.Variables_Set
Bases: object

Wrapper for PPL’s Variables_Set class.
A set of variables’ indexes.

EXAMPLES:

Build the empty set of variable indexes:

```
sage: from sage.libs.ppl import Variable, Variables_Set
sage: Variables_Set()  # Variables_Set of cardinality 0
```

Build the singleton set of indexes containing the index of the variable:

```
sage: v123 = Variable(123)
sage: Variables_Set(v123)  # Variables_Set of cardinality 1
```

Build the set of variables’ indexes in the range from one variable to another variable:

```
sage: v127 = Variable(127)
sage: Variables_Set(v123, v127)  # Variables_Set of cardinality 5
```

OK()
Checks if all the invariants are satisfied.

OUTPUT:
Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Variables_Set
sage: v123 = Variable(123)
sage: S = Variables_Set(v123)
sage: S.OK()  # True
```

14.1. Cython wrapper for the Parma Polyhedra Library (PPL)
ascii_dump()
Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd = 'from sage.libs.ppl import Variable, Variables_Set\n'
sage: sage_cmd += 'v123 = Variable(123)\n'
sage: sage_cmd += 'S = Variables_Set(v123)\n'
sage: sage_cmd += 'S.ascii_dump()\n'
sage: from sage.tests.cmdline import test_executable
sage: (out, err, ret) = test_executable(['sage', '-c', sage_cmd],
        →timeout=100) # long time, indirect doctest
sage: print(err)
# long time
/... DeprecationWarning: The Sage wrappers around ppl are now superseded by
→the standalone pplpy.
Please use import 'ppl' instead of 'sage.libs.ppl'.
See http://trac.sagemath.org/23024 for details.
...
variables( 1 )
123
```

insert(v)
Inserts the index of variable \(v\) into the set.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Variables_Set
sage: S = Variables_Set()
sage: v123 = Variable(123)
sage: S.insert(v123)
sage: S.space_dimension()
124
```

space_dimension()
Returns the dimension of the smallest vector space enclosing all the variables whose indexes are in the set.

OUTPUT:
Integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Variables_Set
sage: v123 = Variable(123)
sage: S = Variables_Set(v123)
sage: S.space_dimension()
124
```

sage.libs.ppl.closure_point(expression=0, divisor=1)
Construc a closure point.

See Generator.closure_point() for documentation.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, closure_point
sage: y = Variable(1)
sage: closure_point(2*y, 5)
closure_point(0/5, 2/5)
```

Sage Reference Manual: C/C++ Library Interfaces, Release 8.7

**sage.libs.ppl.equation**(*expression*)

Construct an equation.

**INPUT:**

- *expression* - a *LinearExpression*.

**OUTPUT:**

The equation *expression* == 0.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, equation
sage: y = Variable(1)
sage: 2*y+1 == 0
2*x1+1==0
sage: equation(2*y+1)
2*x1+1==0
```

**sage.libs.ppl.inequality**(*expression*)

Construct an inequality.

**INPUT:**

- *expression* - a *LinearExpression*.

**OUTPUT:**

The inequality *expression* >= 0.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, inequality
sage: y = Variable(1)
sage: 2*y+1 >= 0
2*x1+1>=0
sage: inequality(2*y+1)
2*x1+1>=0
```

**sage.libs.ppl.line**(*expression*)

Construct a line.

See `Generator.line()` for documentation.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, line
sage: y = Variable(1)
sage: line(2*y)
line(0, 1)
```

**sage.libs.ppl.point**(*expression=0, divisor=1*)

Construct a point.

See `Generator.point()` for documentation.

**EXAMPLES:**

```python
sage: from sage.libs.ppl import Variable, point
sage: y = Variable(1)
sage: point(2*y, 5)
point(0/5, 2/5)
```
sage.libs.ppl.ray(expression)
Construct a ray.

See Generator.ray() for documentation.

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, ray
doctest:...: Python import statement is not pep8 compliant
sage: y = Variable(1)
sage: ray(2*y)
sage: ray(0, 1)
```

sage.libs.ppl.strict_inequality(expression)
Construct a strict inequality.

INPUT:
- expression – a Linear_Expression.

OUTPUT:
The inequality \( \text{expression} > 0 \).

EXAMPLES:

```python
sage: from sage.libs.ppl import Variable, strict_inequality
doctest:...: Python import statement is not pep8 compliant
sage: y = Variable(1)
sage: 2*y+1 > 0
2*x1+1>0
sage: strict_inequality(2*y+1)
2*x1+1>0
```
15.1 Hyperelliptic Curve Point Finding, via ratpoints (deprecated)

This module is deprecated, use PARI instead:

```
sage: pari(EllipticCurve("389a1")).ellratpoints(4)
[-2, 0], [-2, -1], [-1, 1], [-1, -2], [0, 0], [0, -1], [1, 0], [1, -1], [3, 5], [3, -6], [4, 8], [4, -9], [-3/4, 7/8], [-3/4, -15/8]
sage: pari("[x^3 + x^2 - 2*x, 1]").hyperellratpoints(4)
[-2, 0], [-2, -1], [-1, 1], [-1, -2], [0, 0], [0, -1], [1, 0], [1, -1], [3, 5], [3, -6], [4, 8], [4, -9], [-3/4, 7/8], [-3/4, -15/8]
sage.libs.ratpoints.ratpoints(coeffs, H, verbose=False, max=0, min_x_denom=None, max_x_denom=None, intervals=())
```

Access the ratpoints library to find points on the hyperelliptic curve:

\[ y^2 = a_n x^n + \cdots + a_1 x + a_0. \]

**INPUT:**

- `coeffs` – list of integer coefficients \(a_0, a_1, \ldots, a_n\)
- `H` – the bound for the denominator and the absolute value of the numerator of the \(x\)-coordinate
- `verbose` – if True, ratpoints will print comments about its progress
- `max` – maximum number of points to find (if 0, find all of them)

**OUTPUT:**

The points output by this program are points in (1, ceil(n/2), 1)-weighted projective space. If \(n\) is even, then the associated homogeneous equation is \(y^2 = a_n x^n + \cdots + a_1 x z^n + a_0 z^{n+1}\) while if \(n\) is odd, it is \(y^2 = a_n x^n z + \cdots + a_1 x z^n + a_0 z^{n+1}\).

**EXAMPLES:**

```
sage: from sage.libs.ratpoints import ratpoints
doctest:....: DeprecationWarning: the module sage.libs.ratpoints is deprecated;
use pari.ellratpoints or pari.hyperellratpoints instead
See http://trac.sagemath.org/24531 for details.
sage: for x,y,z in ratpoints([1..6], 200):
    print(-1*y^2 + 1*z^6 + 2*x*z^5 + 3*x^2*z^4 + 4*x^3*z^3 + 5*x^4*z^2 +
          6*x^5*z)
0
0
0
0
```

(continues on next page)
The denominator of $x$ can be restricted, for example to find integral points:

```python
sage: from sage.libs.ratpoints import ratpoints
sage: coeffs = [400, -112, 0, 1]
```

```python
def ratpoints(coeffs, 10^n6, max_x_denom=1, intervals=[[-10,0],[1000,2000]])
```

```python
[(1, 0, 0), (-8, 28, 1), (-8, -28, 1), (-7, 29, 1), (-7, -29, 1),
 (-4, 28, 1), (-4, -28, 1), (0, 20, 1), (0, -20, 1), (1368, 50596, 1),
 (1368, -50596, 1), (1624, 65444, 1), (1624, -65444, 1)]
```

```python
def ratpoints(coeffs, 1000, min_x_denom=100, max_x_denom=200)
```

```python
[(1, 0, 0),
 (-656, 426316, 121),
 (-656, -426316, 121),
 (452, 85052, 121),
 (452, -85052, 121),
 (988, 80036, 121),
 (988, -80036, 121),
 (-556, 773188, 169),
 (-556, -773188, 169),
 (264, 432068, 169),
 (264, -432068, 169)]
```

Finding the integral points on the compact component of an elliptic curve:

```python
sage: E = EllipticCurve([0,1,0,-35220,-1346400])
```

```python
def E.division_polynomial(2).roots(multiplicities=False)
```

```python
def coeffs = [E.a6(),E.a4(),E.a2(),1]
```

```python
def ratpoints(coeffs, 1000, max_x_denom=1, intervals=[e3,e2])
```

```python
[(1, 0, 0),
 (-165, 0, 1),
 (-162, 366, 1),
```
(-162, -366, 1),
(-120, 1080, 1),
(-120, -1080, 1),
(-90, 1050, 1),
(-90, -1050, 1),
(-85, 1020, 1),
(-85, -1020, 1),
(-42, 246, 1),
(-42, -246, 1),
(-40, 0, 1)
16.1 Readline

This is the library behind the command line input, it takes keypresses until you hit Enter and then returns it as a string to Python. We hook into it so we can make it redraw the input area.

EXAMPLES:

```python
sage: from sage.libs.readline import *
sage: replace_line('foobar', 0)
sage: set_point(3)
sage: print('current line:', repr(copy_text(0, get_end())))
current line: 'foobar'
sage: print('cursor position:', get_point())
cursor position: 3
```

When printing with `interleaved_output` the prompt and current line is removed:

```python
sage: with interleaved_output():
    ....:    print('output')
    ....:    print('current line: ',
    ....:          repr(copy_text(0, get_end())))
    ....:    print('cursor position:', get_point())
output
current line: ''
cursor position: 0
```

After the interleaved output, the line and cursor is restored to the old value:

```python
sage: print('current line:', repr(copy_text(0, get_end())))
current line: 'foobar'
sage: print('cursor position:', get_point())
cursor position: 3
```

Finally, clear the current line for the remaining doctests:

```python
sage: replace_line('', 1)
```

`sage.libs.readline.clear_signals()`

Remove the readline signal handlers

Remove all of the Readline signal handlers installed by `set_signals()`

EXAMPLES:
sage: from sage.libs.readline import clear_signals
sage: clear_signals()
0

```
sage.libs.readline.copy_text(pos_start, pos_end)
Return a copy of the text between start and end in the current line.

INPUT:
  • pos_start, pos_end – integer. Start and end position.

OUTPUT:
  String.

EXAMPLES:
```
sage: from sage.libs.readline import copy_text, replace_line
sage: replace_line('foobar', 0)
sage: copy_text(1, 5)
'ooba'

```
sage.libs.readline.forced_update_display()
Force the line to be updated and redisplayed, whether or not Readline thinks the screen display is correct.

EXAMPLES:
```
sage: from sage.libs.readline import forced_update_display
sage: forced_update_display()
0

```
sage.libs.readline.get_end()
Get the end position of the current input

OUTPUT:
  Integer

EXAMPLES:
```
sage: from sage.libs.readline import get_end
sage: get_end()
0

```
sage.libs.readline.get_point()
Get the cursor position

OUTPUT:
  Integer

EXAMPLES:
```
sage: from sage.libs.readline import get_point, set_point
sage: get_point()
0
sage: set_point(5)
sage: get_point()
5
sage: set_point(0)
```
sage.libs.readline.initialize()
Initialize or re-initialize Readline’s internal state. It’s not strictly necessary to call this; readline() calls it before reading any input.

EXAMPLES:

```
sage: from sage.libs.readline import initialize
sage: initialize()
0
```

class sage.libs.readline.interleaved_output
Context manager for asynchronous output

This allows you to show output while at the readline prompt. When the block is left, the prompt is restored even if it was clobbered by the output.

EXAMPLES:

```
sage: from sage.libs.readline import interleaved_output
sage: with interleaved_output():
.....:   print('output')
output
```

sage.libs.readline.print_status()
Print readline status for debug purposes

EXAMPLES:

```
sage: from sage.libs.readline import print_status
sage: print_status()
catch_signals: 1
catch_sigwinch: 1
```

sage.libs.readline.replace_line(text, clear_undo)
Replace the contents of rl_line_buffer with text.

The point and mark are preserved, if possible.

INPUT:

- text – the new content of the line.
- clear_undo – integer. If non-zero, the undo list associated with the current line is cleared.

EXAMPLES:

```
sage: from sage.libs.readline import copy_text, replace_line
sage: replace_line('foobar', 0)
sage: copy_text(1, 5)
'ooba'
```

sage.libs.readline.set_point(point)
Set the cursor position

INPUT:


EXAMPLES:
sage: from sage.libs.readline import get_point, set_point
sage: get_point()
0
sage: set_point(5)
sage: get_point()
5
sage: set_point(0)

sage.libs.readline.set_signals()
Install the readline signal handlers

Install Readline’s signal handler for SIGINT, SIGQUIT, SIGTERM, SIGALRM, SIGTSTP, SIGTTIN, SIGTTOU, and SIGWINCH, depending on the values of rl_catch_signals and rl_catch_sigwinch.

EXAMPLES:

sage: from sage.libs.readline import set_signals
sage: set_signals()
0
17.1 Symmetrica library

sage.libs.symmetrica.symmetrica.bdg_symmetrica(part, perm)
Calculates the irreducible matrix representation $D^\text{part}(\text{perm})$, whose entries are of integral numbers.


sage.libs.symmetrica.symmetrica.chartafel_symmetrica(n)
you enter the degree of the symmetric group, as INTEGER object and the result is a MATRIX object: the charactertable of the symmetric group of the given degree.

EXAMPLES:

```python
sage: symmetrica.chartafel(3)
[ 1 1 1]
[-1 0 2]
[ 1 -1 1]
sage: symmetrica.chartafel(4)
[ 1 1 1 1]
[-1 0 -1 1 3]
[ 0 -1 2 0 2]
[ 1 0 -1 -1 3]
[-1 1 1 -1 1]
```

sage.libs.symmetrica.symmetrica.charvalue_symmetrica(irred, cls, table=None)
you enter a PARTITION object part, labelling the irreducible character, you enter a PARTITION object class, labeling the class or class may be a PERMUTATION object, then result becomes the value of that character on that class or permutation. Note that the table may be NULL, in which case the value is computed, or it may be taken from a precalculated charactertable.

FIXME: add table parameter

EXAMPLES:

```python
sage: n = 3
sage: m = matrix([[symmetrica.charvalue(irred, cls) for cls in Partitions(n)] for \n˓→irred in Partitions(n)]); m
[ 1 1 1]
[-1 0 2]
[ 1 -1 1]
sage: m == symmetrica.chartafel(n)
True
sage: n = 4
sage: m = matrix([[symmetrica.charvalue(irred, cls) for cls in Partitions(n)] for \n˓→irred in Partitions(n)]
```

(continues on next page)
computes the expansion of a elementary symmetric function labeled by a INTEGER number as a POLYNOM
erg. The object number may also be a PARTITION or a ELM_SYM object. The INTEGER length specifies the
length of the alphabet. Both routines are the same.

EXAMPLES:

sage: a = symmetrica.compute_elmsym_with_alphabet(2,2); a
x0*x1
sage: a.parent()
Multivariate Polynomial Ring in x0, x1 over Integer Ring
sage: a = symmetrica.compute_elmsym_with_alphabet([2],2); a
x0*x1
sage: symmetrica.compute_elmsym_with_alphabet(3,2)
0
sage: symmetrica.compute_elmsym_with_alphabet([3,2],2)
0

computes the expansion of a homogenous(=complete) symmetric function labeled by a INTEGER number as a
POLYNOM erg. The object number may also be a PARTITION or a HOM_SYM object. The INTEGER laenge
specifies the length of the alphabet. Both routines are the same.

EXAMPLES:

sage: symmetrica.compute_homsym_with_alphabet(3,1,'x')
x^3
sage: symmetrica.compute_homsym_with_alphabet([2,1],1,'x')
x^3
sage: symmetrica.compute_homsym_with_alphabet([2,1],2,'x')
x0^3 + 2*x0^2*x1 + 2*x0*x1^2 + x1^3
sage: symmetrica.compute_homsym_with_alphabet([2,1],2,'a,b')
a^3 + 2*a^2*b + 2*a*b^2 + b^3
sage: symmetrica.compute_homsym_with_alphabet([2,1],2,'x').parent()
Multivariate Polynomial Ring in x0, x1 over Integer Ring

computes the expansion of a monomial symmetric function labeled by a PARTITION number as a POLYNOM
erg. The INTEGER laenge specifies the length of the alphabet.

EXAMPLES:
sage: symmetrica.compute_monomial_with_alphabet([2,1],2,'x')
\text{x0}^2 + \text{x0}\times\text{x1} + \text{x0}\times\text{x1}^2
sage: symmetrica.compute_monomial_with_alphabet([1,1],2,'x')
0
sage: symmetrica.compute_monomial_with_alphabet(2,2,'x')
\text{x0}^2 + \text{x1}^2
sage: symmetrica.compute_monomial_with_alphabet(2,2,'a,b')
a^2 + b^2
sage: symmetrica.compute_monomial_with_alphabet(2,2,'x').parent()
\text{Multivariate Polynomial Ring in x0, x1 over Integer Ring}

sage.libs.symmetrica.symmetrica.\text{compute\_powsym\_with\_alphabet\_symmetrica}(n, length, alphabet='x')

computes the expansion of a power symmetric function labeled by a INTEGER label or by a PARTITION label or a POW\_SYM label as a POLYNOM erg. The INTEGER length specifies the length of the alphabet.

EXAMPLES:

sage: symmetrica.compute_powsym_with_alphabet(2,2,'x')
\text{x0}^2 + \text{x1}^2
sage: symmetrica.compute_powsym_with_alphabet(2,2,'x').parent()
\text{Multivariate Polynomial Ring in x0, x1 over Integer Ring}
sage: symmetrica.compute_powsym_with_alphabet([2],2,'x')
\text{x0}^2 + \text{x1}^2
sage: symmetrica.compute_powsym_with_alphabet([2],2,'a,b')
a^2 + b^2
sage: symmetrica.compute_powsym_with_alphabet([2,1],2,'a,b')
a^3 + a^2\times b + a\times b^2 + b^3

sage.libs.symmetrica.symmetrica.\text{compute\_schur\_with\_alphabet\_det\_symmetrica}(part, length, alphabet='x')

EXAMPLES:

sage: symmetrica.compute_schur_with_alphabet_det(2,2)
\text{x0}^2 + \text{x0}\times\text{x1} + \text{x1}^2
sage: symmetrica.compute_schur_with_alphabet_det([2],2)
\text{x0}^2 + \text{x0}\times\text{x1} + \text{x1}^2
sage: symmetrica.compute_schur_with_alphabet_det(Partition([2]),2)
\text{x0}^2 + \text{x0}\times\text{x1} + \text{x1}^2
sage: symmetrica.compute_schur_with_alphabet_det(Partition([2]),2,'y')
y0^2 + y0\times y1 + y1^2
sage: symmetrica.compute_schur_with_alphabet_det(Partition([2]),2,'a,b')
a^2 + a\times b + b^2

sage.libs.symmetrica.symmetrica.\text{compute\_schur\_with\_alphabet\_symmetrica}(part, length, alphabet='x')

Computes the expansion of a schurfuntion labeled by a partition PART as a POLYNOM erg. The INTEGER length specifies the length of the alphabet.
EXAMPLES:

```
sage: symmetrica.compute_schur_with_alphabet(2,2)
x0^2 + x0*x1 + x1^2
sage: symmetrica.compute_schur_with_alphabet([2],2)
x0^2 + x0*x1 + x1^2
sage: symmetrica.compute_schur_with_alphabet(Partition([2]),2)
x0^2 + x0*x1 + x1^2
sage: symmetrica.compute_schur_with_alphabet(Partition([2]),2,'y')
y0^2 + y0*y1 + y1^2
sage: symmetrica.compute_schur_with_alphabet(Partition([2]),2,'a,b')
a^2 + a*b + b^2
sage: symmetrica.compute_schur_with_alphabet([2,1],1,'x')
0
```

```
sage.libs.symmetrica.symmetrica.dimension_schur_symmetrica(s)
you enter a SCHUR object a, and the result is the dimension of the corresponding representation of the symmetric group sn.
```

```
sage.libs.symmetrica.symmetrica.dimension_symmetrization_symmetrica(n, part)
computes the dimension of the degree of a irreducible representation of the GL_n, n is a INTEGER object, labeled by the PARTITION object a.
```

```
sage.libs.symmetrica.symmetrica.divdiff_perm_schubert_symmetrica(perm, a)
Returns the result of applying the divided difference operator \( \delta_i \) to \( a \) where \( a \) is either a permutation or a Schubert polynomial over QQ.
EXAMPLES:

```
sage: symmetrica.divdiff_perm_schubert([2,3,1], [3,2,1])
X[2, 1]
sage: symmetrica.divdiff_perm_schubert([3,1,2], [3,2,1])
X[1, 3, 2]
sage: symmetrica.divdiff_perm_schubert([3,2,4,1], [3,2,1])
Traceback (most recent call last):
...
ValueError: cannot apply \( \delta_{[3, 2, 4, 1]} \) to a (= [3, 2, 1])
```

```
sage.libs.symmetrica.symmetrica.divdiff_schubert_symmetrica(i, a)
Returns the result of applying the divided difference operator \( \delta_i \) to \( a \) where \( a \) is either a permutation or a Schubert polynomial over QQ.
EXAMPLES:

```
sage: symmetrica.divdiff_schubert(1, [3,2,1])
X[2, 3, 1]
sage: symmetrica.divdiff_schubert(2, [3,2,1])
X[3, 1, 2]
sage: symmetrica.divdiff_schubert(3, [3,2,1])
Traceback (most recent call last):
...
ValueError: cannot apply \( \delta_{3} \) to a (= [3, 2, 1])
```

```
sage.libs.symmetrica.symmetrica.end()
```

```
sage.libs.symmetrica.symmetrica.gupta_nm_symmetrica(n, m)
this routine computes the number of partitions of \( n \) with maximal part \( m \). The result is erg. The input \( n,m \) must be INTEGER objects. The result is freed first to an empty object. The result must be a different from \( m \) and \( n \).
```
sage.libs.symmetrica.symmetrica.gupta_tafel_symmetrica(max)

It computes the table of the above values. The entry n,m is the result of gupta_nm. mat is freed first. max must be an INTEGER object, it is the maximum weight for the partitions. max must be different from result.

sage.libs.symmetrica.symmetrica.hall_littlewood_symmetrica(part)

Computes the so called Hall Littlewood Polynomials, i.e. a SCHUR object, whose coefficient are polynomials in one variable. The method, which is used for the computation is described in the paper: A.O. Morris The Characters of the group GL(n,q) Math Zeitschr 81, 112-123 (1963)

sage.libs.symmetrica.symmetrica.kostka_number_symmetrica(shape, content)

Computes the kostkanumber, i.e. the number of tableaux of given shape, which is a PARTITION object, and of given content, which also is a PARTITION object, or a VECTOR object with INTEGER entries. The result is an INTEGER object, which is freed to an empty object at the beginning. The shape could also be a SKEWPARTITION object, then we compute the number of skewtableaux of the given shape.

EXAMPLES:

```python
sage: symmetrica.kostka_number([2,1],[1,1,1])
2
sage: symmetrica.kostka_number([1,1,1],[1,1,1])
1
sage: symmetrica.kostka_number([3],[1,1,1])
1
```

sage.libs.symmetrica.symmetrica.kostka_tab_symmetrica(shape, content)

Computes the list of tableaux of given shape and content. shape is a PARTITION object or a SKEWPARTITION object and content is a PARTITION object or a VECTOR object with INTEGER entries, the result becomes a LIST object whose entries are the computed TABLEAUX object.

EXAMPLES:

```python
sage: symmetrica.kostka_tab([3],[1,1,1])
[[[1, 2, 3]]]
```

sage.libs.symmetrica.symmetrica.kostka_tafel_symmetrica(n)

Returns the table of Kostka numbers of weight n.

EXAMPLES:

```python
sage: symmetrica.kostka_tafel(1)
[[1]]
```

(continues on next page)
\[
\begin{bmatrix}
1 & 2 & 1 \\
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 \\
1 & 2 & 1 & 1 \\
1 & 3 & 2 & 3 & 1
\end{bmatrix}
\]

sage: symmetrica.kostka_tafel(4)
\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 \\
1 & 2 & 1 & 1 & 0 \\
1 & 3 & 2 & 3 & 1
\end{bmatrix}
\]

sage: symmetrica.kostka_tafel(5)
\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 \\
1 & 2 & 1 & 1 & 0 & 0 \\
1 & 2 & 2 & 1 & 1 & 0 \\
1 & 3 & 3 & 3 & 2 & 1 \\
1 & 4 & 5 & 6 & 5 & 4 & 1
\end{bmatrix}
\]

\begin{verbatim}
sage.libs.symmetrica.symmetrica.kranztafel_symmetrica(a, b)
sage.libs.symmetrica.symmetrica.mult_monomial_monomial_symmetrica(m1, m2)
sage.libs.symmetrica.symmetrica.mult_schubert_schubert_symmetrica(a, b)
sage.libs.symmetrica.symmetrica.mult_schubert_variable_symmetrica(a, i)
\end{verbatim}

**EXAMPLES:**

\begin{verbatim}
sage: {a,b,c} = symmetrica.kranztafel(2,2)
sage: a
\begin{bmatrix}
1 & -1 & 1 & -1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
-1 & 1 & 1 & -1 & 1 \\
0 & 0 & 2 & 0 & -2 \\
-1 & -1 & 1 & 1 & 1
\end{bmatrix}
sage: b
[2, 2, 1, 2, 1]
sage: for m in c: print(m)
[0 0]
[0 1]
[0 0]
[1 0]
[0 2]
[0 0]
[1 1]
[0 0]
[2 0]
[0 0]
\end{verbatim}

\begin{verbatim}
sage: symmetrica.mult_schubert_schubert([3,2,1], [3,2,1])
X[5, 3, 1, 2, 4]
sage: symmetrica.mult_schubert_variable_symmetrica(3, 1)
\end{verbatim}

\begin{verbatim}
sage.libs.symmetrica.symmetrica.mult_schubert_variable_symmetrica(a, i)
\end{verbatim}

**EXAMPLES:**

\begin{verbatim}
sage: symmetrica.mult_schubert_schubert([3,2,1], [3,2,1])
X[5, 3, 1, 2, 4]
sage: symmetrica.mult_schubert_variable_symmetrica(3, 1)
\end{verbatim}
EXAMPLES:

```python
sage: symmetrica.mutl_schur_schur_symmetric(a, b)
```

sage.libs.symmetrica.symmetrica.mutl_schur_schur_symmetric(a, b)

sage.libs.symmetrica.symmetrica.ndg_symmetrica(part, perm)

computes the decomposition of a schubertpolynomial labeled by the permutation perm, as a sum of Schurfun-
tion. FIXME!

sage.libs.symmetrica.symmetrica.odd_to_strict_part_symmetrica(part)

implements the bijection between partitions with odd parts and strict partitions. input is a VECTOR type parti-
tion, the result is a partition of the same weight with different parts.

sage.libs.symmetrica.symmetrica.odg_symmetrica(part, perm)

Calculates the irreducible matrix representation D^part(perm), which consists of real numbers.


sage.libs.symmetrica.symmetrica.outerproduct_schur_symmetrica(parta, partb)

you enter two PARTITION objects, and the result is a SCHUR object, which is the expansion of the product
of the two schurfunctions, labbeled by the two PARTITION objects partida and partb. Of course this can also be
interpreted as the decomposition of the outer tensor product of two irreducibe representations of the symmetric
group.

EXAMPLES:

```python
sage: symmetrica.outerproduct_schur([2],[2])
```

sage.libs.symmetrica.symmetrica.part_part_skewscher_symmetrica(outer, inner)

Return the skew Schur function s_.{outer/inner}.

EXAMPLES:

```python
sage: symmetrica.part_part_skewscher([3,2,1],[2,1])
s[1, 1, 1] + 2*s[2, 1] + s[3]
```

sage.libs.symmetrica.symmetrica.plethysm_symmetrica(outer, inner)

sage.libs.symmetrica.symmetrica.q_core_symmetrica(part, d)

computes the q-core of a PARTITION object part. This is the remaining partition (=res) after removing of all
hooks of length d (= INTEGER object). The result may be an empty object, if the whole partition disappears.

sage.libs.symmetrica.symmetrica.random_partition_symmetrica(n)

Return a random partition p of the entered weight w.

w must be an INTEGER object, p becomes a PARTITION object. Type of partition is VECTOR . It uses the
algorithm of Nijenhuis and Wilf, p.76

sage.libs.symmetrica.symmetrica.scalarproduct_schubert_symmetrica(a, b)

EXAMPLES:
sage: symmetrica.scalarproduct_schubert([3,2,1], [3,2,1])
X[1, 3, 5, 2, 4]
sage: symmetrica.scalarproduct_schubert([3,2,1], [2,1,3])
X[1, 2, 4, 3]

Calculates the irreducible matrix representation \(D^\nu\) of the same weight with only odd parts.

sage: symmetrica.scalarproduct_schur_symmetrica(s1, s2)
sage: symmetrica.schur_schur_plet_symmetrica(outer, inner)


sage: symmetrica.specht_dg_symmetrica(part, perm)
sage: symmetrica.start()

sage: symmetrica.strict_to_odd_part_symmetrica(part)

Converts a symmetric polynomial with base ring QQ or ZZ into a symmetric function in the elementary basis.

sage: symmetrica.t_ELMSYM_HOMSYM_symmetrica(elsyn)
sage: symmetrica.t_ELMSYM_MONOMIAL_symmetrica(elsyn)
sage: symmetrica.t_ELMSYM_POWSYM_symmetrica(elsyn)
sage: symmetrica.t_ELMSYM_SCHUR_symmetrica(elsyn)

Converts a multivariate polynomial \(a\) to a Schubert polynomial.

EXAMPLES:

sage: R.<x1,x2,x3> = QQ[]
sage: w0 = x1^2*x2
sage: symmetrica.t_POLYNOM_SCHUBERT_symmetrica(w0)
X[3, 2, 1]
sage.libs.symmetrica.symmetrica.t_POLYNOM_SCHUR_symmetrica(p)
    Converts a symmetric polynomial with base ring QQ or ZZ into a symmetric function in the Schur basis.

sage.libs.symmetrica.symmetrica.t_POWSYM_ELMSYM_symmetrica(powsym)

sage.libs.symmetrica.symmetrica.t_POWSYM_HOMSYM_symmetrica(powsym)

sage.libs.symmetrica.symmetrica.t_POWSYM_MONOMIAL_symmetrica(powsym)

sage.libs.symmetrica.symmetrica.t_POWSYM_SCHUR_symmetrica(powsym)

sage.libs.symmetrica.symmetrica.t_SCHUBERT_POLYNOM_symmetrica(a)
    Converts a Schubert polynomial to a ‘regular’ multivariate polynomial.

    EXAMPLES:

    sage: symmetrica.t_SCHUBERT_POLYNOM([3,2,1])
    x0^2*x1

sage.libs.symmetrica.symmetrica.t_SCHUR_ELMSYM_symmetrica(schur)

sage.libs.symmetrica.symmetrica.t_SCHUR_HOMSYM_symmetrica(schur)

sage.libs.symmetrica.symmetrica.t_SCHUR_MONOMIAL_symmetrica(schur)

sage.libs.symmetrica.symmetrica.t_SCHUR_POWSYM_symmetrica(schur)

sage.libs.symmetrica.symmetrica.test_integer(x)
    Tests functionality for converting between Sage’s integers and symmetrica’s integers.

    EXAMPLES:

    sage: from sage.libs.symmetrica.symmetrica import test_integer
    sage: test_integer(1)
    1
    sage: test_integer(-1)
    -1
    sage: test_integer(2^33)
    8589934592
    sage: test_integer(-2^33)
    -8589934592
    sage: test_integer(2^100)
    1267650600228229401496703205376
    sage: test_integer(-2^100)
    -1267650600228229401496703205376
    sage: for i in range(100):
    ....:     if test_integer(2^i) != 2^i:
    ....:         print("Failure at {}".format(i))

17.1. Symmetrica library
INDICES AND TABLES

- Index
- Module Index
- Search Page
sage.libs.ecl, 3
sage.libs.eclib.constructor, 34
sage.libs.eclib.homspace, 31
sage.libs.eclib.interface, 11
sage.libs.eclib.mat, 28
sage.libs.eclib.mwrank, 26
sage.libs.eclib.newforms, 29
sage.libs.flint.arith, 39
sage.libs.flint.flint, 37
sage.libs.flint.fmpz_poly, 37
sage.libs.gap.context_managers, 75
sage.libs.gap.element, 84
sage.libs.gap.gap_functions, 76
sage.libs.gap.libgap, 77
sage.libs.gap.saved_workspace, 98
sage.libs.gap.test, 83
sage.libs.gap.test_long, 76
sage.libs.gap.util, 76
sage.libs.gsl.array, 45
sage.libs.lcalc.lcalc_Lfunction, 47
sage.libs.libecm, 43
sage.libs.linbox.linbox_flint_interface, 101
sage.libs.lrcalc.lrcalc, 103
sage.libs.mpmath.utils, 111
sage.libsntl.all, 117
sage.libs.pari, 119
sage.libs.pari.convert_sage, 121
sage.libs.ppl, 127
sage.libs.ratpoints, 189
sage.libs.readline, 193
sage.libs.singular.function, 55
sage.libs.singular.function_factory, 63
sage.libs.singular.groebner_strategy, 70
sage.libs.singular.option, 64
sage.libs.singular.polynomial, 64
sage.libs.singular.ring, 69
sage.libs.singular.singular, 63
sage.libs.symmetrica.symmetrica, 197

r
sage.rings.pari_ring, 125
INDEX

A
add_constraint() (sage.libs.ppl.MIP_Problem method), 151
add_constraint() (sage.libs.ppl.Polyhedron method), 162
add_constraints() (sage.libs.ppl.MIP_Problem method), 152
add_constraints() (sage.libs.ppl.Polyhedron method), 163
add_generator() (sage.libs.ppl.Polyhedron method), 163
add_generators() (sage.libs.ppl.Polyhedron method), 164
add_scalar() (sage.libs.eclib.mat.Matrix method), 28
add_space_dimensions_and_embed() (sage.libs.ppl.MIP_Problem method), 152
add_space_dimensions_and_embed() (sage.libs.ppl.Polyhedron method), 165
add_space_dimensions_and_project() (sage.libs.ppl.Polyhedron method), 166
add_to_integer_space_dimensions() (sage.libs.ppl.MIP_Problem method), 152
affine_dimension() (sage.libs.ppl.Polyhedron method), 166
ainvs() (sage.libs.eclib.interface.mwrank_EllipticCurve method), 12
all_homogeneous_terms_are_zero() (sage.libs.ppl.Linear_Expression method), 148
all_vectors() (in module sage.libs.singular.function), 59
all_vectors() (in module sage.libs.singular.function), 59
ascii_dump() (sage.libs.ppl.Constraint method), 131
ascii_dump() (sage.libs.ppl.Constraint_System method), 136
ascii_dump() (sage.libs.ppl-generator method), 139
ascii_dump() (sage.libs.ppl.Polyhedron method), 146
ascii_dump() (sage.libs.ppl.Linear_Expression method), 149
ascii_dump() (sage.libs.ppl.Poly_Con_Relation method), 158
ascii_dump() (sage.libs.ppl.Poly_Gen_Relation method), 160
ascii_dump() (sage.libs.ppl.Polyhedron method), 167
ascii_dump() (sage.libs.ppl.Variables_Set method), 185
atomp() (sage.libs.ecl.EclObject method), 4

B
BaseCallHandler (class in sage.libs.singular.function), 56
bdg_symmetrica() (in module sage.libs.symmetrict.symmetricta), 197
bell_number() (in module sage.libs.flib.arith), 39
bernoulli_number() (in module sage.libs.flib.arith), 39
bitcount() (in module sage.libs.mpmath.utils), 111
bounds_from_above() (sage.libs.ppl.Polyhedron method), 167
bounds_from_below() (sage.libs.ppl.Polyhedron method), 168
<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_Polyhedron (class in sage.libs.ppl), 129</td>
</tr>
<tr>
<td>caar() (sage.libs.ecl.EclObject method), 5</td>
</tr>
<tr>
<td>cadr() (sage.libs.ecl.EclObject method), 5</td>
</tr>
<tr>
<td>call() (in module sage.libs.mpmath.utils), 111</td>
</tr>
<tr>
<td>car() (sage.libs.ecl.EclObject method), 5</td>
</tr>
<tr>
<td>cdar() (sage.libs.ecl.EclObject method), 6</td>
</tr>
<tr>
<td>cddr() (sage.libs.ecl.EclObject method), 6</td>
</tr>
<tr>
<td>cdr() (sage.libs.ecl.EclObject method), 6</td>
</tr>
<tr>
<td>certain() (sage.libs.eclib.interface.mwrank_EllipticCurve method), 12</td>
</tr>
<tr>
<td>characteristic() (sage.libs.singular.function.RingWrap method), 57</td>
</tr>
<tr>
<td>characteristic() (sage.rings.pari_ring.PariRing method), 125</td>
</tr>
<tr>
<td>characterp() (sage.libs.eclib.interface.mwrank_EllipticCurve method), 12</td>
</tr>
<tr>
<td>charpoly() (sage.libs.eclib.mat.Matrix method), 28</td>
</tr>
<tr>
<td>chartafel_symmetrica() (in module sage.libs.symmetrica.symmetrica), 197</td>
</tr>
<tr>
<td>charvalue_symmetrica() (in module sage.libs.symmetrica.symmetrica), 197</td>
</tr>
<tr>
<td>clear() (sage.libs.ppl.Constraint_System method), 136</td>
</tr>
<tr>
<td>clear() (sage.libs.ppl.Generator_System method), 146</td>
</tr>
<tr>
<td>clear() (sage.libs.ppl.MIP_Problem method), 153</td>
</tr>
<tr>
<td>clear_signals() (in module sage.libs.readline), 193</td>
</tr>
<tr>
<td>closure_point() (in module sage.libs.ppl), 186</td>
</tr>
<tr>
<td>closure_point() (sage.libs.ppl.Generator method), 139</td>
</tr>
<tr>
<td>coefficient() (sage.libs.ppl.Constraint method), 131</td>
</tr>
<tr>
<td>coefficient() (sage.libs.ppl.Generator method), 140</td>
</tr>
<tr>
<td>coefficient() (sage.libs.ppl.Linear_Expression method), 149</td>
</tr>
<tr>
<td>coefficients() (sage.libs.ppl.Constraint method), 132</td>
</tr>
<tr>
<td>coefficients() (sage.libs.ppl.Generator method), 140</td>
</tr>
<tr>
<td>coefficients() (sage.libs.ppl.Linear_Expression method), 149</td>
</tr>
<tr>
<td>collect() (sage.libs.gap.libgap.Gap method), 80</td>
</tr>
<tr>
<td>compute_elmsym_with_alphabet_symmetrica() (in module sage.libs.symmetrica.symmetrica), 198</td>
</tr>
<tr>
<td>compute_homsym_with_alphabet_symmetrica() (in module sage.libs.symmetrica.symmetrica), 198</td>
</tr>
<tr>
<td>compute_monomial_with_alphabet_symmetrica() (in module sage.libs.symmetrica.symmetrica), 198</td>
</tr>
<tr>
<td>compute_powsym_with_alphabet_symmetrica() (in module sage.libs.symmetrica.symmetrica), 199</td>
</tr>
<tr>
<td>compute_rank() (sage.libs.lcalc.lcalc_Lfunction.Lfunction method), 47</td>
</tr>
<tr>
<td>compute_schur_with_alphabet_det_symmetrica() (in module sage.libs.symmetrica.symmetrica), 199</td>
</tr>
<tr>
<td>compute_schur_with_alphabet_symmetrica() (in module sage.libs.symmetrica.symmetrica), 199</td>
</tr>
<tr>
<td>concatenate_assign() (sage.libs.ppl.Polyhedron method), 168</td>
</tr>
<tr>
<td>conductor() (sage.libs.eclib.interface.mwrank_EllipticCurve method), 13</td>
</tr>
<tr>
<td>cons() (sage.libs.ecl.EclObject method), 7</td>
</tr>
<tr>
<td>consp() (sage.libs.ecl.EclObject method), 7</td>
</tr>
<tr>
<td>constrains() (sage.libs.ppl.Polyhedron method), 169</td>
</tr>
<tr>
<td>Constraint (class in sage.libs.ppl), 130</td>
</tr>
<tr>
<td>Constraint_System (class in sage.libs.ppl), 135</td>
</tr>
<tr>
<td>Constraint_System_iterator (class in sage.libs.ppl), 138</td>
</tr>
<tr>
<td>constraints() (sage.libs.ppl.Polyhedron method), 169</td>
</tr>
<tr>
<td>contains() (sage.libs.ppl.Polyhedron method), 170</td>
</tr>
<tr>
<td>contains_integer_point() (sage.libs.ppl.Polyhedron method), 170</td>
</tr>
<tr>
<td>Converter (class in sage.libs.singular.function), 56</td>
</tr>
<tr>
<td>coprod() (in module sage.libs.lrcalc.lrcalc), 105</td>
</tr>
</tbody>
</table>
copy_text() (in module sage.libs.readline), 194
count_GAP_objects() (sage.libs.gap.libgap.Gap method), 80
CPS_height_bound() (sage.libs.eclib.interface.mwrank_EllipticCurve method), 12
CremonaModularSymbols() (in module sage.libs.eclib.constructor), 34
currRing_wrapper() (in module sage.libs.singular.constructor), 69

dedekind_sum() (in module sage.libs.flint.arith), 40
depc() (sage.libs.gap.element.GapElement method), 84
degree() (sage.libs.fmpz_poly.Fmpz_poly method), 37
derivative() (sage.libs.fmpz_poly.Fmpz_poly method), 37
difference_assign() (sage.libs.ppl.Polyhedron method), 171
dimension() (sage.libs.eclib.homspace.ModularSymbols method), 31
dimension_schur_symmetrica() (in module sage.libs.symmetrica.symmetrica), 200
div_diff_schubert_symmetrica() (in module sage.libs.symmetrica.symmetrica), 200
div_diff_schubert_symmetrica() (in module sage.libs.symmetrica.symmetrica), 200
divisor() (sage.libs.ppl.Generator method), 141
drop_some_non_integer_points() (sage.libs.ppl.Polyhedron method), 172

ecl_eval() (in module sage.libs.ecl), 9
EclListIterator (class in sage.libs.ecl), 3
EclObject (class in sage.libs.ecl), 3
ecmfactor() (in module sage.libs.libecm), 43
ECModularSymbol (class in sage.libs.eclib.newforms), 29
Element (sage.libs.gap.libgap.Gap attribute), 80
Element (sage.rings.pari_ring.PariRing attribute), 125
empty() (sage.libs.ppl.Constraint_System method), 136
empty() (sage.libs.ppl.Generator_System method), 146
equation() (in module sage.libs.symmetrica.symmetrica), 200
euler() (sage.libs.ecl.EclObject method), 7
eval() (sage.libs.gap.libgap.Gap method), 80
evaluate_objective_function() (sage.libs.ppl.MIP_Problem method), 153

find_zeros() (sage.libs.lcalc.lcalc.Lfunction.Lfunction method), 47
find_zeros_via_N() (sage.libs.lcalc.lcalc.Lfunction.Lfunction method), 48
fixnump() (sage.libs.mpmath.mpmath.EclObject method), 7
Fmpz_poly (class in sage.libs.fmpz_poly), 37
forced_update_display() (in module sage.libs.readline), 194
free_flint_stack() (in module sage.libs.fmpz_poly), 37
from_man_exp() (in module sage.libs.mpmath.mpmathutils), 112
function_factory() (sage.libs.gap.libgap.Gap method), 80

Gap (class in sage.libs.gap.libgap), 79
gap_root() (in module sage.libs.gap.util), 76
GapElement (class in sage.libs.gap.element), 84
GapElement_Boolean (class in sage.libs.gap.element), 87
GapElement_Cyclotomic (class in sage.libs.gap.element), 87
GapElement_FiniteField (class in sage.libs.gap.element), 88
GapElement_Float (class in sage.libs.gap.element), 89
GapElement_Function (class in sage.libs.gap.element), 90
GapElement_Integer (class in sage.libs.gap.element), 90
GapElement_IntegerMod (class in sage.libs.gap.element), 90
GapElement_List (class in sage.libs.gap.element), 91
GapElement_MethodProxy (class in sage.libs.gap.element), 93
GapElement_Permutation (class in sage.libs.gap.element), 94
GapElement_Rational (class in sage.libs.gap.element), 94
GapElement_Record (class in sage.libs.gap.element), 95
GapElement_RecordIterator (class in sage.libs.gap.element), 96
GapElement_Ring (class in sage.libs.gap.element), 96
GapElement_String (class in sage.libs.gap.element), 98
GAPError, 76
gen_to_sage() (in module sage.libs.pari.convert_sage), 121
Generator (class in sage.libs.ppl), 138
Generator_System (class in sage.libs.ppl), 145
Generator_System_iterator (class in sage.libs.ppl), 147
generators() (sage.libs.ppl.Polyhedron method), 172
gens() (sage.libs.eclib.interface.mwrank_EllipticCurve method), 13
get_end() (in module sage.libs.readline), 194
get_global() (sage.libs.gap.libgap.Gap method), 81
get_owned_objects() (in module sage.libs.gap.util), 77
get_point() (in module sage.libs.readline), 194
get_precision() (in module sage.libs.eclib.mwrank), 26
global_context() (sage.libs.gap.libgap.Gap method), 81
GlobalVariableContext (class in sage.libs.gap.context_managers), 75
GroebnerStrategy (class in sage.libs.singular.groebner_strategy), 70
GSLDoubleArray (class in sage.libs.gsl.array), 45
gupta_nm_symmetrica() (in module sage.libs.symmetrica.symmetrica), 200
gupta_tafel_symmetrica() (in module sage.libs.symmetrica.symmetrica), 200
hall_littlewood_symmetrica() (in module sage.libs.symmetrica.symmetrica), 201
hardy_z_function() (sage.libs.lcalc.lcalc_Lfunction.Lfunction method), 49
harmonic_number() (in module sage.libs.flint.arith), 40
has_equalities() (sage.libs.ppl.Constraint_System method), 137
has_strict_inequalities() (sage.libs.ppl.Constraint_System method), 137
hecke_matrix() (sage.libs.eclib.homspace.ModularSymbols method), 32
id() (sage.libs.ppl.Variable method), 184
ideal() (sage.libs.singular.groebner_strategy.GroebnerStrategy method), 71
ideal() (sage.libs.singular.groebner_strategy.NCGroebnerStrategy method), 71
implies() (sage.libs.ppl.Poly_Con_Relation method), 158
implies() (sage.libs.ppl.Poly_Gen_Relation method), 160
inequality() (in module sage.libs.ppl), 187
inhomogeneous_term() (sage.libs.ppl.Constraint method), 132
inhomogeneous_term() (sage.libs.ppl.Linear_Expression method), 150
init_ecl() (in module sage.libs.ecl), 9
initialize() (in module sage.libs.readline), 194
initprimes() (in module sage.libs.eclib.mwrank), 27
insert() (sage.libs.ppl.Constraint_System method), 137
insert() (sage.libs.ppl.Generator_System method), 147
insert() (sage.libs.ppl.Variables_Set method), 186
interleaved_output (class in sage.libs.readline), 195
intersection_assign() (sage.libs.ppl.Polyhedron method), 173
is_bool() (sage.libs.gap.element.GapElement method), 84
is_bounded() (sage.libs.ppl.Polyhedron method), 173
is_C_int() (sage.libs.gap.element.GapElement_Integer method), 90
is_closure_point() (sage.libs.ppl.Generator method), 141
is_commutative() (sage.libs.singular.function.RingWrap method), 57
is_cuspidal() (sage.libs.eclib.homspace.ModularSymbols method), 32
is_discrete() (sage.libs.ppl.Polyhedron method), 173
is_disjoint() (sage.libs.ppl.Poly_Con_Relation method), 159
is_disjoint_from() (sage.libs.ppl.Polyhedron method), 174
is_empty() (sage.libs.ppl.Polyhedron method), 174
is_equality() (sage.libs.ppl.Constraint method), 133
is_equivalent_to() (sage.libs.ppl.Constraint method), 133
is_field() (sage.rings.pari_ring.PariRing method), 125
is_function() (sage.libs.gap.element.GapElement method), 85
is_included() (sage.libs.ppl.Poly_Con_Relation method), 159
is_inconsistent() (sage.libs.ppl.Constraint method), 133
is_inequality() (sage.libs.ppl.Constraint method), 133
is_line() (sage.libs.ppl.Generator method), 142
is_line_or_ray() (sage.libs.ppl.Generator method), 142
is_list() (sage.libs.gap.element.GapElement method), 85
is_nonstrict_inequality() (sage.libs.ppl.Constraint method), 134
is_permutation() (sage.libs.gap.element.GapElement method), 85
is_point() (sage.libs.ppl.Generator method), 142
is_ray() (sage.libs.ppl.Generator method), 143
is_record() (sage.libs.gap.element.GapElement method), 85
is_sage_wrapper_for_singular_ring() (in module sage.libs.singular.function), 59
is_satisfiable() (sage.libs.ppl.MIP_Problem method), 153
is_singular_poly_wrapper() (in module sage.libs.singular.function), 59
is_strict_inequality() (sage.libs.ppl.Constraint method), 134
is_string() (sage.libs.gap.element.GapElement method), 86
is_tautological() (sage.libs.ppl.Constraint method), 134
is_topologically_closed() (sage.libs.ppl.Polyhedron method), 175
is_universe() (sage.libs.ppl.Polyhedron method), 175
is_zero() (sage.libs.ppl.Linear_Expression method), 150
isogeny_class() (sage.libs.eclib.interface.mwrank.EllipticCurve method), 13
isqrt() (in module sage.libs.mpmath.utils), 113
K

KernelCallHandler (class in sage.libs.singular.function), 56
kostka_number_symmetrica() (in module sage.libs.symmetrica.symmetrica), 201
kostka_tab_symmetrica() (in module sage.libs.symmetrica.symmetrica), 201
kostka_tafel_symmetrica() (in module sage.libs.symmetrica.symmetrica), 201
kranztafel_symmetrica() (in module sage.libs.symmetrica.symmetrica), 202

L

left_shift() (sage.libs.flint.fmpz_poly.Fmpz_poly method), 38
level() (sage.libs.eclib.homspace.ModularSymbols method), 33
Lfunction (class in sage.libs.lcalc.lcalc_Lfunction), 47
Lfunction_C (class in sage.libs.lcalc.lcalc_Lfunction), 50
Lfunction_D (class in sage.libs.lcalc.lcalc_Lfunction), 51
Lfunction_from_character() (in module sage.libs.lcalc.lcalc_Lfunction), 52
Lfunction_from_elliptic_curve() (in module sage.libs.lcalc.lcalc_Lfunction), 52
Lfunction_I (class in sage.libs.lcalc.lcalc_Lfunction), 51
Lfunction_Zeta (class in sage.libs.lcalc.lcalc_Lfunction), 52
lib() (in module sage.libs.singular.function), 59
LibraryCallHandler (class in sage.libs.singular.function), 56
LibSingularOptions (class in sage.libs.singular.option), 65
LibSingularOptions_abstract (class in sage.libs.singular.option), 68
LibSingularOptionsContext (class in sage.libs.singular.option), 67
LibSingularVerboseOptions (class in sage.libs.singular.option), 68
lift() (sage.libs.gap.element.GapElement_FiniteField method), 88
lift() (sage.libs.gap.element.GapElement_IntegerMod method), 91
line() (in module sage.libs.ppl), 187
line() (sage.libs.ppl.Generator method), 143
Linear_Expression (class in sage.libs.ppl), 147
list() (sage.libs.flint.fmpz_poly.Fmpz_poly method), 38
list_of_functions() (in module sage.libs.singular.function), 60
listp() (sage.libs.ecl.EclObject method), 8
load() (sage.libs.singular.option.LibSingularOptions_abstract method), 68
lrcoef() (in module sage.libs.lrcalc.lrcalc), 106
lrcoefUnsafe() (in module sage.libs.lrcalc.lrcalc), 106
lrskew() (in module sage.libs.lrcalc.lrcalc), 107

M

Matrix (class in sage.libs.eclib.mat), 28
matrix() (sage.libs.gap.element.GapElement_List method), 92
MatrixFactory (class in sage.libs.eclib.mat), 29
max_space_dimension() (sage.libs.ppl.Polyhedron method), 175
maximize() (sage.libs.ppl.Polyhedron method), 176
mem() (sage.libs.gap.libgap.Gap method), 81
minimize() (sage.libs.ppl.Polyhedron method), 176
minimized_constraints() (sage.libs.ppl.Polyhedron method), 177
minimizedGenerators() (sage.libs.ppl.Polyhedron method), 178
MIP_Problem (class in sage.libs.ppl), 150
ModularSymbols (class in sage.libs.eclib.homspace), 31
mpmath_to_sage() (in module sage.libs.mpmath.utils), 113
mult() (in module sage.libs.lrcalc.lrcalc), 107
mult_monomial_monomial_symmetrica() (in module sage.libs.symmetrica.symmetrica), 202
mult_schubert() (in module sage.libs.lrcalc.lrcalc), 108
mult_schubert_schubert_symmetrica() (in module sage.libs.symmetrica.symmetrica), 202
mult_schubert_variable_symmetrica() (in module sage.libs.symmetrica.symmetrica), 202
mult_schur_schur_symmetrica() (in module sage.libs.symmetrica.symmetrica), 203
mwrack_EllipticCurve (class in sage.libs.eclib.interface), 11
mwrack_MordellWeil (class in sage.libs.eclib.interface), 17

N
NCGroebnerStrategy (class in sage.libs.singular.groebner_strategy), 71
ncols() (sage.libs.eclib.mat.Matrix method), 28
ndg_symmetrica() (in module sage.libs.symmetrica.symmetrica), 203
newtrans_symmetrica() (in module sage.libs.symmetrica.symmetrica), 203
next() (sage.libs.ecl.EclListIterator method), 3
next() (sage.libs.gap.element.GapElement_RecordIterator method), 96
next() (sage.libs.ppl.Constraint_System_iterator method), 138
next() (sage.libs.ppl.Generator_System_iterator method), 147
ngens() (sage.libs.singular.function.RingWrap method), 57
NNC_Polyhedron (class in sage.libs.ppl), 156
normal_form() (sage.libs.singular.groebner_strategy.GroebnerStrategy method), 71
normal_form() (sage.libs.singular.groebner_strategy.NCGroebnerStrategy method), 72
normalize() (in module sage.libs.mpmath.utils), 114
nothing() (sage.libs.ppl.Poly_Con_Relation method), 159
nothing() (sage.libs.ppl.Poly_Gen_Relation method), 161
npars() (sage.libs.singular.function.RingWrap method), 57
nrows() (sage.libs.eclib.mat.Matrix method), 29
nullp() (sage.libs.ecl.EclObject method), 8
number_of_cusps() (sage.libs.eclib.homspace.ModularSymbols method), 33
number_of_partitions() (in module sage.libs.flint.arith), 40

O
objective_function() (sage.libs.ppl.MIP_Problem method), 154
ObjWrapper (class in sage.libs.gap.util), 76
odd_to_strict_part_symmetrica() (in module sage.libs.symmetrica.symmetrica), 203
odg_symmetrica() (in module sage.libs.symmetrica.symmetrica), 203
OK() (sage.libs.ppl.Constraint method), 131
OK() (sage.libs.ppl.Constraint_System method), 136
OK() (sage.libs.ppl.Generator method), 139
OK() (sage.libs.ppl.Generator_System method), 145
OK() (sage.libs.ppl.Linear_Expression method), 148
OK() (sage.libs.ppl.MIP_Problem method), 151
OK() (sage.libs.ppl.Poly_Con_Relation method), 159
OK() (sage.libs.ppl.Poly_Gen_Relation method), 160
OK() (sage.libs.ppl.Polyhedron method), 161
OK() (sage.libs.ppl.Variable method), 184
OK() (sage.libs.ppl.Variables_Set method), 185
one() (sage.libs.gap.libgap.Gap method), 81
opt (sage.libs.singular.option.LibSingularOptionsContext attribute), 68
optimal_value() (sage.libs.ppl.MIP_Problem method), 154
optimization_mode() (sage.libs.ppl.MIP_Problem method), 154
optimizing_point() (sage.libs.ppl.MIP_Problem method), 155
ordering_string() (sage.libs.singular.function.RingWrap method), 57
outerproduct_schur_symmetrica() (in module sage.libs.symmetrica.symmetrica), 203

P
par_names() (sage.libs.singular.function.RingWrap method), 58
Pari (class in sage.rings.pari_ring), 125
PariRing (class in sage.rings.pari_ring), 125
part_part_skewschur_symmetrica() (in module sage.libs.symmetrica.symmetrica), 203
plethysm_symmetrica() (in module sage.libs.symmetrica.symmetrica), 203
point() (in module sage.libs.ppl), 187
point() (sage.libs.ppl.Generator method), 143
points() (sage.libs.eclib.interface.mwrank_MordellWeil method), 20
poison_currRing() (in module sage.libs.singular.ring), 69
Poly_Con_Relation (class in sage.libs.ppl), 157
poly_difference_assign() (sage.libs.ppl.Polyhedron method), 178
Poly_Gen_Relation (class in sage.libs.ppl), 160
poly_hull_assign() (sage.libs.ppl.Polyhedron method), 179
Polyhedron (class in sage.libs.ppl), 161
pow_truncate() (sage.libs.flint.fmpz_poly.Fmpz_poly method), 38
print_currRing() (in module sage.libs.singular.ring), 70
print_objects() (in module sage.libs.ecl), 9
print_status() (in module sage.libs.readline), 195
process() (sage.libs.eclib.interface.mwrank_MordellWeil method), 20
pseudo_div() (sage.libs.flint.fmpz_poly.Fmpz_poly method), 39
pseudo_div_rem() (sage.libs.flint.fmpz_poly.Fmpz_poly method), 39
python() (sage.libs.ecl.EclObject method), 8

Q
q_core_symmetrica() (in module sage.libs.symmetrica.symmetrica), 203

R
random_element() (sage.rings.pari_ring.PariRing method), 125
random_partition_symmetrica() (in module sage.libs.symmetrica.symmetrica), 203
rank() (sage.libs.eclib.interface.mwrank_EllipticCurve method), 13
rank() (sage.libs.eclib.interface.mwrank_MordellWeil method), 22
rank_bound() (sage.libs.eclib.interface.mwrank_EllipticCurve method), 14
ratpoints() (in module sage.libs.ratpoints), 189
ray() (in module sage.libs.ppl), 187
ray() (sage.libs.ppl.Generator method), 144
record_name_to_index() (sage.libs.gap.element.GapElement_Record method), 95
regulator() (sage.libs.eclib.interface.mwrank_EllipticCurve method), 14
regulator() (sage.libs.eclib.interface.mwrank_MordellWeil method), 22
relation_with() (sage.libs.ppl.Polyhedron method), 180
remove_higher_space_dimensions() (sage.libs.ppl.Polyhedron method), 181
replace_line() (in module sage.libs.readline), 195
reset_default() (sage.libs.singular.option.LibSingularOptions method), 67
reset_default() (sage.libs.singular.option.LibSingularVerboseOptions method), 69
Resolution (class in sage.libs.singular.function), 56
right_shift() (sage.libs.flint.fmpz_poly.Fmpz_poly method), 39
ring() (sage.libs.singular.function.Converter method), 56
ring() (sage.libs.singular.groebner_strategy.GroebnerStrategy method), 71
ring() (sage.libs.singular.groebner_strategy.NCGroebnerStrategy method), 72
ring_cyclotomic() (sage.libs.gap.element.GapElement_Ring method), 96
ring_finite_field() (sage.libs.gap.element.GapElement_Ring method), 97
ring_integer() (sage.libs.gap.element.GapElement_Ring method), 97
ring_integer_mod() (sage.libs.gap.element.GapElement_Ring method), 97
ring_polynomial() (sage.libs.gap.element.GapElement_Ring method), 97
ring_rational() (sage.libs.gap.element.GapElement_Ring method), 97
ring_wrapper_Py (class in sage.libs.singular.ring), 70
RingWrap (class in sage.libs.singular.function), 56
rplaca() (sage.libs.ecl.EclObject method), 8
rplacd() (sage.libs.ecl.EclObject method), 8

S

sage() (sage.libs.gap.element.GapElement method), 86
sage() (sage.libs.gap.element.GapElement_Boolean method), 87
sage() (sage.libs.gap.element.GapElement_Cyclotomic method), 88
sage() (sage.libs.gap.element.GapElement_FiniteField method), 89
sage() (sage.libs.gap.element.GapElement_Float method), 89
sage() (sage.libs.gap.element.GapElement_Integer method), 90
sage() (sage.libs.gap.element.GapElement_IntegerMod method), 91
sage() (sage.libs.gap.element.GapElement_List method), 93
sage() (sage.libs.gap.element.GapElement_Permutation method), 94
sage() (sage.libs.gap.element.GapElement_Rational method), 94
sage() (sage.libs.gap.element.GapElement_Record method), 96
sage() (sage.libs.gap.element.GapElement_Ring method), 97
sage() (sage.libs.gap.element.GapElement_String method), 98
sage.libs.ecl (module), 3
sage.libs.eclib.constructor (module), 34
sage.libs.eclib.homspace (module), 31
sage.libs.eclib.interface (module), 11
sage.libs.eclib.mat (module), 28
sage.libs.eclib.mwrank (module), 26
sage.libs.eclib.newforms (module), 29
sage.libs.flint.arith (module), 39
sage.libs.flint.fmpz_poly (module), 37
sage.libs.gap.context_managers (module), 75
sage.libs.gap.element (module), 84
sage.libs.gap.gap_functions (module), 76
sage.libs.gap.libgap (module), 77
sage.libs.gap.saved_workspace (module), 98
sage.libs.gap.test (module), 83
sage.libs.gap.test_long (module), 76
sage.libs.gap.util (module), 76
sage.libs.gsl.array (module), 45
sage.libs.lcalc.lcalc_Lfunction (module), 47
sage.libs.libecm (module), 43
sage.libs.linbox.linbox_flint_interface (module), 101

Index 219
sage.libs.lrcalc.lrcalc (module), 103
sage.libs.mpmath.utils (module), 111
sage.libs.ntl.all (module), 117
sage.libs.pari (module), 119
sage.libs.pari.convert_sage (module), 121
sage.libs.ppl (module), 127
sage.libs.ratpoints (module), 189
sage.libs.readline (module), 193
sage.libs.singular.function (module), 55
sage.libs.singular.function_factory (module), 63
sage.libs.singular.groebner_strategy (module), 70
sage.libs.singular.option (module), 64
sage.libs.singular.polynomial (module), 64
sage.libs.singular.ring (module), 69
sage.libs.singular.singular (module), 63
sage.libs.symmetrica.symmetrica (module), 197
sage.rings.pari_ring (module), 125
sage_matrix_over_ZZ() (sage.libs.eclib.mat.Matrix method), 29
sage_to_mpmath() (in module sage.libs.mpmath.utils), 114
saturate() (sage.libs.eclib.interface.mwrank_EllipticCurve method), 14
saturate() (sage.libs.eclib.interface.mwrank_MordellWeil method), 23
saturates() (sage.libs.ppl.Poly_Con_Relation method), 159
save() (sage.libs.singular.option.LibSingularOptions_abstract method), 68
scalarproduct_shubert_symmetrica() (in module sage.libs.symmetrica.symmetrica), 203
scalarproduct_shur_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
schur_schur_plet_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
sdg_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
search() (sage.libs.eclib.interface.mwrank_MordellWeil method), 25
selmer_rank() (sage.libs.eclib.interface.mwrank_EllipticCurve method), 15
set_global() (sage.libs.gap.libgap.Gap method), 82
set_objective_function() (sage.libs.ppl.MIP_Problem method), 155
set_optimization_mode() (sage.libs.ppl.MIP_Problem method), 155
set_point() (in module sage.libs.readline), 195
set_precision() (in module sage.libs.eclib.mwrank), 27
set_seed() (sage.libs.gap.libgap.Gap method), 82
set_signals() (in module sage.libs.readline), 196
set_verbosity() (sage.libs.eclib.interface.mwrank_EllipticCurve method), 16
show() (sage.libs.gap.libgap.Gap method), 82
shutdown_ecl() (in module sage.libs.ecl), 10
sign() (sage.libs.eclib.homspace.ModularSymbols method), 33
silverman_bound() (sage.libs.eclib.interface.mwrank_EllipticCurve method), 16
singular_function() (in module sage.libs.singular.function), 60
SingularFunction (class in sage.libs.singular.function), 58
SingularFunctionFactory (class in sage.libs.singular.function_factory), 63
SingularKernelFunction (class in sage.libs.singular.function), 58
SingularLibraryFunction (class in sage.libs.singular.function), 58
skew() (in module sage.libs.lrcalc.lrcalc), 109
solve() (sage.libs.ppl.MIP_Problem method), 155
space_dimension() (sage.libs.ppl.Constraint method), 135
space_dimension() (sage.libs.ppl.Constraint_System method), 138
space_dimension() (sage.libs.ppl.Generator method), 144
space_dimension() (sage.libs.ppl.Generator_System method), 147
space_dimension() (sage.libs.ppl.Linear_Expression method), 150
space_dimension() (sage.libs.ppl.MIP_Problem method), 156
space_dimension() (sage.libs.ppl.Polyhedron method), 181
space_dimension() (sage.libs.ppl.Variable method), 185
space_dimension() (sage.libs.ppl.Variables_Set method), 186
sparse_hecke_matrix() (sage.libs.eclib.homspace.ModularSymbols method), 33
specht_dg_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
start() (in module sage.libs.symmetrica.symmetrica), 204
str() (sage.libs.eclib.mat.Matrix method), 29
strict_inequality() (in module sage.libs.ppl), 188
strict_to_odd_part_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
strictly_contains() (sage.libs.ppl.Polyhedron method), 181
strictly_intersects() (sage.libs.ppl.Poly_Con_Relation method), 159
subsumes() (sage.libs.ppl.Poly_Gen_Relation method), 161
symbolp() (sage.libs.ecl.EclObject method), 9

t_ELMSYM_HOMSYM_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_ELMSYM_MONOMIAL_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_ELMSYM_POWSYM_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_ELMSYM_SCHUR_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_HOMSYM_ELMSYM_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_HOMSYM_MONOMIAL_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_HOMSYM_POWSYM_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_HOMSYM_SCHUR_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_MONOMIAL_ELMSYM_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_MONOMIAL_HOMSYM_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_MONOMIAL_POWSYM_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_MONOMIAL_SCHUR_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_POLYNOM_ELMSYM_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_POLYNOM_MONOMIAL_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_POLYNOM_POWER_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_POLYNOM_SCHUBERT_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_POLYNOM_SCHUR_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_POLYNOM_SCHUR_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_POLYNOM_SCHUR_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_POLYNOM_SCHUR_symmetrica() (in module sage.libs.symmetrica.symmetrica), 204
t_SCHUBERT_POLYNOM_symmetrica() (in module sage.libs.symmetrica.symmetrica), 205
t_SCHUR_ELMSYM_symmetrica() (in module sage.libs.symmetrica.symmetrica), 205
t_SCHUR_HOMSYM_symmetrica() (in module sage.libs.symmetrica.symmetrica), 205
t_SCHUR_MONOMIAL_symmetrica() (in module sage.libs.symmetrica.symmetrica), 205
t_SCHUR_POWSYM_symmetrica() (in module sage.libs.symmetrica.symmetrica), 205
test_ecl_options() (in module sage.libs.ecl), 10
test_integer() (in module sage.libs.symmetrica.symmetrica), 205
test_iterable_to_vector() (in module sage.libs.lrcalc.lrcalc), 109
test_loop_1() (in module sage.libs.gap.test_long), 76
test_loop_2() (in module sage.libs.gap.test_long), 76
test_loop_3() (in module sage.libs.gap.test_long), 76
 test_sigint_before_ecl_sig_on() (in module sage.libs.ecl), 10
 test_skewtab_to_SkewTableau() (in module sage.libs.lrcalc.lrcalc), 109
 test_write_to_file() (in module sage.libs.gap.test), 83
 timestamp() (in module sage.libs.gap.saved_workspace), 98
 topological_closure_assign() (sage.libs.ppl.Polyhedron method), 182
 trait_names() (sage.libs.singular.function_factory.SingularFunctionFactory method), 63
 truncate() (sage.libs.flint.fmpz_poly.Fmpz_poly method), 39
 two_descent() (sage.libs.clib.interface.mwrank_EllipticCurve method), 17
 type() (sage.libs.ppl.Constraint method), 135
 type() (sage.libs.ppl.Generator method), 145

U
 unconstrain() (sage.libs.ppl.Polyhedron method), 182
 unpickle_GroebnerStrategy0() (in module sage.libs.singular.groebner_strategy), 72
 unpickle_NCGroebnerStrategy0() (in module sage.libs.singular.groebner_strategy), 72
 unset_global() (sage.libs.gap.libgap.Gap method), 83
 upper_bound_assign() (sage.libs.ppl.Polyhedron method), 183

V
 value() (sage.libs.lcalc.lcalc_Lfunction.Lfunction method), 49
 var_names() (sage.libs.singular.function.RingWrap method), 58
 Variable (class in sage.libs.ppl), 183
 Variables_Set (class in sage.libs.ppl), 185
 vector() (sage.libs.gap.element.GapElement_List method), 93

W
 workspace() (in module sage.libs.gap.saved_workspace), 99

Z
 zero() (sage.libs.gap.libgap.Gap method), 83
 zeta() (sage.rings.pari_ring.PariRing method), 126