Sage Reference Manual: C/C++ Library Interfaces

Release 6.3

The Sage Development Team

August 11, 2014

CONTENTS

An underlying philosophy in the development of Sage is that it should provide unified library-level access to the some of the best GPL'd C/C++ libraries. Currently Sage provides some access to MWRANK, NTL, PARI, and Hanke, each of 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., `expect`), since everything is linked together and run as a single process. This is much more robust and efficient than using `expect`.

Each of these interfaces is used by other parts of Sage. For example, mwrank 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.

CHAPTER

ONE

LIBGAP SHARED LIBRARY INTERFACE TO GAP

libGAP shared library Interface to GAP

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

EXAMPLES:

```
sage: a = libgap(10)
sage: a
10
sage: type(a)
<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:

```
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:

```
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:

```
sage: libgap('List([1..10], i->i^2)')
"List([1..10], i->i^2)"
sage: type(_)
<type 'sage.libs.gap.element.GapElement_String'>
```

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

```
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 = libgap.AlternatingGroup(4).GeneratorsOfGroup().sage()
sage: generators   # a Sage list of Sage permutations!
[(1,2,3), (2,3,4)]
sage: PermutationGroup(generators).cardinality()   # computed in Sage
12
sage: libgap.AlternatingGroup(4).Size()            # computed in GAP
12
```

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']
Sym( [ 1 .. 3 ] )
sage: dict(rec)
{'a': 123, 'Sym3': Sym( [ 1 .. 3 ] ), 'b': 456}
```

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:

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

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:

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

1.1 Using the libGAP C library from Cython

The lower-case `libgap_foobar` functions are ones that we added to make the libGAP C shared library. The `libGAP_foobar` methods are the original GAP methods simply prefixed with the string `libGAP_`. The latter were originally not designed to be in a library, so some care needs to be taken to call them.

In particular, you must call `libgap_mark_stack_bottom()` in every function that calls into the libGAP C functions. The reason is that the GAP memory manager will automatically keep objects alive that are referenced in local (stack-allocated) variables. While convenient, this requires to look through the stack to find anything that looks like an address to a memory bag. But this requires vigilance against the following pattern:

```
cdef f()
  libgap_mark_stack_bottom()
  libGAP_function()

cdef g()
  libgap_mark_stack_bottom();
  f()                #  f() changed the stack bottom marker
  libGAP_function()  #  boom
```

The solution is to re-order `g()` to first call `f()`. In order to catch this error, it is recommended that you wrap calls into libGAP in `libgap_enter` / `libgap_exit` blocks and not call `libgap_mark_stack_bottom` manually. So instead, always write

> **cdef f()** libgap_enter() libGAP_function() libgap_exit()
>
> **cdef g()** f() libgap_enter() libGAP_function() libgap_exit()

If you accidentally call `libgap_enter()` twice then an error message is printed to help you debug this:

```
sage: from sage.libs.gap.util import error_enter_libgap_block_twice
sage: error_enter_libgap_block_twice()
Traceback (most recent call last):
...
RuntimeError: Entered a critical block twice
```

AUTHORS:

- William Stein, Robert Miller (2009-06-23): first version
- Volker Braun, Dmitrii Pasechnik, Ivan Andrus (2011-03-25, Sage Days 29): almost complete rewrite; first usable version.
- Volker Braun (2012-08-28, GAP/Singular workshop): update to gap-4.5.5, make it ready for public consumption.

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:

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

TESTS:

```
sage: TestSuite(libgap).run(skip=['_test_category', '_test_elements', '_test_pickling'])
```

`Element`

alias of `GapElement`

`collect``()`

Manually run the garbage collector

EXAMPLES:

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

`count_GAP_objects``()`

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

OUTPUT:

An integer

EXAMPLES:

```
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:

```
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:

```
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:

```
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):
...
ValueError: libGAP: 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:

```
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 libGAP memory usage

The GAP workspace is partitioned into 5 pieces (see gasman.c in the GAP sources for more details):

- The **masterpointer area** contains all the masterpointers of the bags.
- The **old bags area** contains the bodies of all the bags that survived at least one garbage collection. This area is only scanned for dead bags during a full garbage collection.
- The **young bags area** contains the bodies of all the bags that have been allocated since the last garbage collection. This area is scanned for dead bags during each garbage collection.
- The **allocation area** is the storage that is available for allocation of new bags. When a new bag is allocated the storage for the body is taken from the beginning of this area, and this area is correspondingly reduced. If the body does not fit in the allocation area a garbage collection is performed.
- The **unavailable area** is the free storage that is not available for allocation.

OUTPUT:

This function returns a tuple containing 5 integers. Each is the size (in bytes) of the five partitions of the workspace. This will potentially change after each GAP garbage collection.

EXAMPLES:

```
sage: libgap.collect()
sage: libgap.mem()   # random output
(1048576, 6706782, 0, 960930, 0)

sage: libgap.FreeGroup(3)
<free group on the generators [ f1, f2, f3 ]>
sage: libgap.mem()   # random output
(1048576, 6706782, 47571, 913359, 0)

sage: libgap.collect()
sage: libgap.mem()   # random output
(1048576, 6734785, 0, 998463, 0)
```

set_global(*variable*, *value*)

Set a GAP global variable

INPUT:

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

EXAMPLES:

```
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):
...
ValueError: libGAP: Error, VAL_GVAR: No value bound to FooBar
```

show()

Print statistics about the GAP owned object list

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:

```
sage: a = libgap(123)
sage: b = libgap(456)
sage: c = libgap(789)
sage: del b
sage: libgap.show() # random output
11 LibGAP elements currently alive
rec( full := rec( cumulative := 122, deadbags := 9,
deadkb := 0, freekb := 7785, livebags := 304915,
livekb := 47367, time := 33, totalkb := 68608 ),
nfull := 3, npartial := 14 )
```

trait_names()

Return all Gap function names.

OUTPUT:

A list of strings.

EXAMPLES:

```
sage: len(libgap.trait_names()) > 1000
True
```

unset_global(*variable*)

Remove a GAP global variable

INPUT:

- `variable` – string. The variable name.

EXAMPLES:

```
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):
...
ValueError: libGAP: Error, VAL_GVAR: No value bound to FooBar
```

zero_element()

Return (integer) zero in GAP.

OUTPUT:

A `GapElement`.

EXAMPLES:

```
sage: libgap.zero_element()
0
```

LIBGAP ELEMENT WRAPPER

libGAP element wrapper

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

class `sage.libs.gap.element.`**`GapElement`**

Bases: `sage.structure.element.RingElement`

Wrapper for all Gap objects.

Note: In order to create `GapElements` 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:

```
sage: libgap(0)
0
```

If Gap finds an error while evaluating, a corresponding assertion is raised:

```
sage: libgap.eval('1/0')
Traceback (most recent call last):
...
ValueError: libGAP: Error, Rational operations: <divisor> must not be zero
```

Also, a `ValueError` is raised if the input is not a simple expression:

```
sage: libgap.eval('1; 2; 3')
Traceback (most recent call last):
...
ValueError: can only evaluate a single statement
```

`is_bool``()`

Return whether the wrapped GAP object is a GAP boolean.

OUTPUT:

Boolean.

EXAMPLES:

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

`is_function``()`

Return whether the wrapped GAP object is a function.

OUTPUT:

Boolean.

EXAMPLES:

```
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: 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: 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
```

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:

```
sage: m = libgap.eval('[[Z(2^2), Z(2)^0],[0*Z(2), Z(2^2)^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]
```

GAP is also starting to introduce a specialized matrix type. Currently, you need to use `Unpack` to convert it back to a list-of-lists:

```
sage: M = libgap.eval('SL(2,GF(5))').GeneratorsOfGroup()[1]
sage: type(M)       # not a GAP list
<type 'sage.libs.gap.element.GapElement'>
sage: M.IsMatrix()
true
sage: M.matrix()
[4 1]
[4 0]
```

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

```
sage: type(_)
<type 'bool'>

sage: libgap('this is a string').sage()
'this is a string'
sage: type(_)
<type 'str'>
```

trait_names()

Return all Gap function names.

OUTPUT:

A list of strings.

EXAMPLES:

```
sage: x = libgap(1)
sage: len(x.trait_names()) > 1000
True
```

vector(*ring=None*)

Return the list as a vector.

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

OUTPUT:

A Sage vector.

EXAMPLES:

```
sage: m = libgap.eval('[0*Z(2), Z(2^2), Z(2)^0, Z(2^2)^2]');  m
[ 0*Z(2), Z(2^2), Z(2)^0, Z(2^2)^2 ]
sage: vector(m)
(0, a, 1, a + 1)
sage: vector(GF(4,'B'), m)
(0, B, 1, B + 1)
```

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(_)
<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
```

TESTS:

Check that trac ticket #15204 is fixed:

```
sage: libgap.E(3).sage(ring=UniversalCyclotomicField())
E(3)
sage: libgap.E(3).sage(ring=CC)
-0.500000000000000 + 0.866025403784439*I
```

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 finite field has incompatible size
```

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 rational numbers.

EXAMPLES:

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

`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.

OUTPUT:

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
```

TESTS:

```
sage: large = libgap.eval('2^130');  large
1361129467683753853853498429727072845824
sage: large.sage()
1361129467683753853853498429727072845824

sage: huge = libgap.eval('10^9999');  huge     # gap abbreviates very long ints
<integer 100...000 (10000 digits)>
sage: huge.sage().ndigits()
10000
```

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: type(_)
<type 'sage.libs.gap.element.GapElement_Integer'>
```

`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 ringa 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:

```
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`:

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

Range checking is performed:

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

sage(***kwds*)

Return the Sage equivalent of the `GapElement`

OUTPUT:

A Python list.

EXAMPLES:

```
sage: libgap([ 1, 3, 4 ]).sage()
[1, 3, 4]
sage: all( x in ZZ for x in _ )
True
```

class `sage.libs.gap.element.`**`GapElement_MethodProxy`**

Bases: `sage.libs.gap.element.GapElement_Function`

Helper class returned by `GapElement.__getattr__`.

Derived class of 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.

EXAMPLES:

```
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:

```
sage: perm = libgap.eval('(1,5,2)(4,3,8)')
sage: type(perm)
<type 'sage.libs.gap.element.GapElement_Permutation'>
```

`sage``()`

Return the Sage equivalent of the `GapElement`

EXAMPLES:

```
sage: perm_gap = libgap.eval('(1,5,2)(4,3,8)');  perm_gap
(1,5,2)(3,8,4)
sage: perm_gap.sage()
(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:

```
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.

OUTPUT:

A Sage rational number.

EXAMPLES:

```
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'>
```

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

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

Derived class of GapElement for GAP records.

EXAMPLES:

```
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`:

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

Range checking is performed:

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

record_name_to_index(*py_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(!)

EXAMPLE:

```
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()
{'a': 123,
 'Sym3': NotImplementedError('cannot construct equivalent Sage object',),
 '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: list(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_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
```

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()

Return the Sage equivalent of the `GapElement`

OUTPUT:

A Python list.

EXAMPLES:

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

```
sage: s.sage()
'string'
sage: type(_)
<type 'str'>
```

CHAPTER

THREE

FLINT FMPZ_POLY CLASS WRAPPER

FLINT fmpz_poly class wrapper

AUTHORS:

- Robert Bradshaw (2007-09-15) Initial version.
- William Stein (2007-10-02) update for new flint; add arithmetic and creation of coefficients of arbitrary size.

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:

```
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([1,2])
sage: f.pow_truncate(10,3)
3  1 20 180
sage: f.pow_truncate(1000,3)
3  1 2000 1998000
```

pseudo_div(*other*)

pseudo_div_rem(*other*)

right_shift(*n*)

Right shift self by n.

EXAMPLES:

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

truncate(*n*)

Return the truncation of self at degree n.

EXAMPLES:

```
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
```

CHAPTER

FOUR

FPLLL LIBRARY

fpLLL library

Wrapper for the fpLLL library by Damien Stehle, Xavier Pujol and David Cade found at http://perso.ens-lyon.fr/damien.stehle/fplll/.

This wrapper provides access to fpLLL's LLL, BKZ and enumeration implementations.

AUTHORS:

- Martin Albrecht (2007-10) initial release
- Martin Albrecht (2014-03) update to fpLLL 4.0 interface

class `sage.libs.fplll.fplll.`**`FP_LLL`**

Bases: `object`

A basic wrapper class to support conversion to/from Sage integer matrices and executing LLL/BKZ computations.

`BKZ`(*block_size*, *delta='LLL_DEF_DELTA'*, *float_type=None*, *precision=0*, *max_loops=0*, *max_time=0*, *verbose=False*, *no_lll=False*, *bounded_lll=False*, *auto_abort=False*)

Run BKZ reduction.

INPUT:

- `block_size` – an integer from 1 to `nrows`
- `delta` – (default: `0.99`) LLL parameter $0.25 < \delta < 1.0$
- `float_type` – (default: `None`) can be one of the following:
  - `None` - for automatic choice
  - `'double'`
  - `'long double'`
  - `'dpe'`
  - `'mpfr'`
- `verbose` – (default: `False`) be verbose
- `no_lll` – (default: `False`) to use LLL
- `bounded_lll` – (default: `False`) bounded LLL
- `precision` – (default: `0` for automatic choice) bit precision to use if `fp` is `'rr'`
- `max_loops` – (default: `0` for no restriction) maximum number of full loops
- `max_time` – (default: `0` for no restricion) stop after time seconds (up to loop completion)

- `auto_abort` – (default: `False`) heuristic, stop when the average slope of $\log(\|b_i^*\|)$ does not decrease fast enough

OUTPUT:

Nothing is returned but the internal state is modified.

EXAMPLES:

```
sage: from sage.libs.fplll.fplll import FP_LLL
sage: A = sage.crypto.gen_lattice(type='random', n=1, m=60, q=2^90, seed=42)
sage: F = FP_LLL(A)

sage: F.LLL()
sage: F._sage_()[0].norm().n()
7.810...

sage: F.BKZ(10)
sage: F._sage_()[0].norm().n()
6.164...
```

HKZ()

Run HKZ reduction.

OUTPUT:

Nothing is returned but the internal state is modified.

EXAMPLE:

```
sage: from sage.libs.fplll.fplll import FP_LLL
sage: A = sage.crypto.gen_lattice(type='random', n=1, m=10, q=2^60, seed=42)
sage: F = FP_LLL(A)
sage: F.HKZ()
sage: F._sage_()
[ -8  27   7  19  10  -5  14  34   4 -18]
[-22  23   3 -14  11  30 -12  26  17  26]
[-20   6 -18  33 -26  16   8 -15 -14 -26]
[ -2  30   9 -30 -28 -19  -7 -28  12 -15]
[-16   1  25 -23 -11 -21 -39   4 -34 -13]
[-27  -2 -24 -67  32 -13  -6   0  15  -4]
[  9 -12   7  31  22  -7 -63  11  27  36]
[ 14  -4   0 -21 -17  -7  -9  35  79 -22]
[-17 -16  54  21   0 -17  28 -45  -6  12]
[ 43  16   6  30  24  17 -39 -46 -18 -22]
```

LLL(*delta='LLL_DEF_DELTA'*, *eta='LLL_DEF_ETA'*, *method=None*, *float_type=None*, *precision=0*, *verbose=False*, *siegel=False*, *early_red=False*)

(δ, η)-LLL reduce this lattice.

INPUT:

- `delta` – (default: `0.99`) parameter $0.25 < \delta < 1.0$
- `eta `` -- (default: ``0.51`) parameter $0.5 \leq \eta < \sqrt{\delta}$
- `method` – (default: `None`) can be one of the following:
  - `'wrapper'` (`None`)
  - `'proved'`
  - `'fast'`

  – `'heuristic'`

- `float_type` – (default: `None`) can be one of the following:
  – `None` - for automatic choice
  – `'double'`
  – `'long double'`
  – `'dpe'`
  – `'mpfr'`
- `precision` – (default: `0` for automatic choice) precision to use
- `verbose` – (default: `False`) be verbose
- `siegel` – (default: `False`) use Siegel conditioning
- `early_red` – (default: `False`) use early reduction

OUTPUT:

Nothing is returned but the internal state is modified.

EXAMPLES:

```
sage: from sage.libs.fplll.fplll import FP_LLL
sage: A = random_matrix(ZZ,10,10); A
[   -8     2     0     0     1    -1     2     1   -95    -1]
[   -2   -12     0     0     1    -1     1    -1    -2    -1]
[    4    -4    -6     5     0     0    -2     0     1    -4]
[   -6     1    -1     1     1    -1     1    -1    -3     1]
[    1     0     0    -3     2    -2     0    -2     1     0]
[   -1     1     0     0     1    -1     4    -1     1    -1]
[   14     1    -5     4    -1     0     2     4     1     1]
[   -2    -1     0     4    -3     1    -5     0    -2    -1]
[   -9    -1    -1     3     2     1    -1     1    -2     1]
[   -1     2    -7     1     0     2     3 -1955   -22    -1]

sage: F = FP_LLL(A)
sage: F.LLL(method="wrapper")
sage: L = F._sage_(); L
[   1    0    0   -3    2   -2    0   -2    1    0]
[  -1    1    0    0    1   -1    4   -1    1   -1]
[  -2    0    0    1    0   -2   -1   -3    0   -2]
[  -2   -2    0   -1    3    0   -2    0    2    0]
[   1    1    1    2    3   -2   -2    0    3    1]
[  -4    1   -1    0    1    1    2    2   -3    3]
[   1   -3   -7    2    3   -1    0    0   -1   -1]
[   1   -9    1    3    1   -3    1   -1   -1    0]
[   8    5   19    3   27    6   -3    8  -25  -22]
[ 172  -25   57  248  261  793   76 -839  -41  376]
sage: L.is_LLL_reduced()
True
sage: L.hermite_form() == A.hermite_form()
True

sage: set_random_seed(0)
sage: A = random_matrix(ZZ,10,10); A
[   -8     2     0     0     1    -1     2     1   -95    -1]
[   -2   -12     0     0     1    -1     1    -1    -2    -1]
[    4    -4    -6     5     0     0    -2     0     1    -4]
[   -6     1    -1     1     1    -1     1    -1    -3     1]
[    1     0     0    -3     2    -2     0    -2     1     0]
[   -1     1     0     0     1    -1     4    -1     1    -1]
[   14     1    -5     4    -1     0     2     4     1     1]
[   -2    -1     0     4    -3     1    -5     0    -2    -1]
[   -9    -1    -1     3     2     1    -1     1    -2     1]
[   -1     2    -7     1     0     2     3 -1955   -22    -1]

sage: F = FP_LLL(A)
sage: F.LLL(method="proved")
sage: L = F._sage_(); L
[   1    0    0   -3    2   -2    0   -2    1    0]
[  -1    1    0    0    1   -1    4   -1    1   -1]
[  -2    0    0    1    0   -2   -1   -3    0   -2]
[  -2   -2    0   -1    3    0   -2    0    2    0]
[   1    1    1    2    3   -2   -2    0    3    1]
[  -4    1   -1    0    1    1    2    2   -3    3]
[   1   -3   -7    2    3   -1    0    0   -1   -1]
[   1   -9    1    3    1   -3    1   -1   -1    0]
[   8    5   19    3   27    6   -3    8  -25  -22]
[ 172  -25   57  248  261  793   76 -839  -41  376]

sage: L.is_LLL_reduced()
True
sage: L.hermite_form() == A.hermite_form()
True

sage: A = random_matrix(ZZ,10,10,x=-(10^5),y=10^5)
sage: f = FP_LLL(A)
sage: f.LLL(method="fast")
sage: L = f._sage_()
sage: L.is_LLL_reduced(eta=0.51,delta=0.99)
True
sage: L.hermite_form() == A.hermite_form()
True

sage: set_random_seed(0)
sage: A = random_matrix(ZZ,10,10); A
[   -8     2     0     0     1    -1     2     1   -95    -1]
[   -2   -12     0     0     1    -1     1    -1    -2    -1]
[    4    -4    -6     5     0     0    -2     0     1    -4]
[   -6     1    -1     1     1    -1     1    -1    -3     1]
[    1     0     0    -3     2    -2     0    -2     1     0]
[   -1     1     0     0     1    -1     4    -1     1    -1]
[   14     1    -5     4    -1     0     2     4     1     1]
[   -2    -1     0     4    -3     1    -5     0    -2    -1]
[   -9    -1    -1     3     2     1    -1     1    -2     1]
[   -1     2    -7     1     0     2     3 -1955   -22    -1]

sage: F = FP_LLL(A)
sage: F.LLL(method="fast", early_red=True)
sage: L = F._sage_(); L
[   1    0    0   -3    2   -2    0   -2    1    0]
[  -1    1    0    0    1   -1    4   -1    1   -1]
[  -2    0    0    1    0   -2   -1   -3    0   -2]
[  -2   -2    0   -1    3    0   -2    0    2    0]
[   1    1    1    2    3   -2   -2    0    3    1]
[  -4    1   -1    0    1    1    2    2   -3    3]
[   1   -3   -7    2    3   -1    0    0   -1   -1]
[   1   -9    1    3    1   -3    1   -1   -1    0]
[   8    5   19    3   27    6   -3    8  -25  -22]
[ 172  -25   57  248  261  793   76 -839  -41  376]

sage: L.is_LLL_reduced(eta=0.51,delta=0.99)
True
sage: L.hermite_form() == A.hermite_form()
True

sage: set_random_seed(0)
sage: A = random_matrix(ZZ,10,10); A
[   -8     2     0     0     1    -1     2     1   -95    -1]
[   -2   -12     0     0     1    -1     1    -1    -2    -1]
[    4    -4    -6     5     0     0    -2     0     1    -4]
[   -6     1    -1     1     1    -1     1    -1    -3     1]
[    1     0     0    -3     2    -2     0    -2     1     0]
[   -1     1     0     0     1    -1     4    -1     1    -1]
[   14     1    -5     4    -1     0     2     4     1     1]
[   -2    -1     0     4    -3     1    -5     0    -2    -1]
[   -9    -1    -1     3     2     1    -1     1    -2     1]
[   -1     2    -7     1     0     2     3 -1955   -22    -1]

sage: F = FP_LLL(A)
sage: F.LLL(method="heuristic")
sage: L = F._sage_(); L
[   1    0    0   -3    2   -2    0   -2    1    0]
[  -1    1    0    0    1   -1    4   -1    1   -1]
[  -2    0    0    1    0   -2   -1   -3    0   -2]
[  -2   -2    0   -1    3    0   -2    0    2    0]
[   1    1    1    2    3   -2   -2    0    3    1]
[  -4    1   -1    0    1    1    2    2   -3    3]
[   1   -3   -7    2    3   -1    0    0   -1   -1]
[   1   -9    1    3    1   -3    1   -1   -1    0]
[   8    5   19    3   27    6   -3    8  -25  -22]
[ 172  -25   57  248  261  793   76 -839  -41  376]

sage: L.is_LLL_reduced(eta=0.51,delta=0.99)
True
sage: L.hermite_form() == A.hermite_form()
True

sage: set_random_seed(0)
sage: A = random_matrix(ZZ,10,10); A
[   -8     2     0     0     1    -1     2     1   -95    -1]
[   -2   -12     0     0     1    -1     1    -1    -2    -1]
[    4    -4    -6     5     0     0    -2     0     1    -4]
[   -6     1    -1     1     1    -1     1    -1    -3     1]
[    1     0     0    -3     2    -2     0    -2     1     0]
[   -1     1     0     0     1    -1     4    -1     1    -1]
[   14     1    -5     4    -1     0     2     4     1     1]
[   -2    -1     0     4    -3     1    -5     0    -2    -1]
[   -9    -1    -1     3     2     1    -1     1    -2     1]
[   -1     2    -7     1     0     2     3 -1955   -22    -1]

sage: F = FP_LLL(A)
sage: F.LLL(method="heuristic", early_red=True)
sage: L = F._sage_(); L
```

```
[   1    0    0   -3    2   -2    0   -2    1    0]
[  -1    1    0    0    1   -1    4   -1    1   -1]
[  -2    0    0    1    0   -2   -1   -3    0   -2]
[  -2   -2    0   -1    3    0   -2    0    2    0]
[   1    1    1    2    3   -2   -2    0    3    1]
[  -4    1   -1    0    1    1    2    2   -3    3]
[   1   -3   -7    2    3   -1    0    0   -1   -1]
[   1   -9    1    3    1   -3    1   -1   -1    0]
[   8    5   19    3   27    6   -3    8  -25  -22]
[ 172  -25   57  248  261  793   76 -839  -41  376]

sage: L.is_LLL_reduced(eta=0.51,delta=0.99)
True
sage: L.hermite_form() == A.hermite_form()
True
```

fast (*precision=0*, *eta=0.51*, *delta=0.99*, *implementation=None*)

Perform LLL reduction using fpLLL's fast implementation. This implementation is the fastest floating point implementation currently available in the free software world.

INPUT:

- `precision` – (default: auto) internal precision
- `eta` – (default: `0.51`) LLL parameter η with $1/2 \le \eta < \sqrt{\delta}$
- `delta` – (default: `0.99`) LLL parameter δ with $1/4 < \delta \le 1$
- `implementation` – ignored

OUTPUT:

Nothing is returned but the internal is state modified.

EXAMPLE:

```
sage: from sage.libs.fplll.fplll import FP_LLL
sage: A = random_matrix(ZZ,10,10,x=-(10^5),y=10^5)
sage: f = FP_LLL(A)
sage: f.fast()
sage: L = f._sage_()
sage: L.is_LLL_reduced(eta=0.51,delta=0.99)
True
sage: L.hermite_form() == A.hermite_form()
True
```

fast_early_red (*precision=0*, *eta=0.51*, *delta=0.99*, *implementation=None*)

Perform LLL reduction using fpLLL's fast implementation with early reduction.

This implementation inserts some early reduction steps inside the execution of the 'fast' LLL algorithm. This sometimes makes the entries of the basis smaller very quickly. It occurs in particular for lattice bases built from minimal polynomial or integer relation detection problems.

INPUT:

- `precision` – (default: auto) internal precision
- `eta` – (default: `0.51`) LLL parameter η with $1/2 \le \eta < \sqrt{\delta}$
- `delta` – (default: `0.99`) LLL parameter δ with $1/4 < \delta \le 1$
- `implementation` – (default: `"mpfr"`) which floating point implementation to use, can be one of the following:

–"double"

–"dpe"

–"mpfr"

OUTPUT:

Nothing is returned but the internal state modified.

EXAMPLE:

```
sage: from sage.libs.fplll.fplll import FP_LLL
sage: A = random_matrix(ZZ,10,10); A
[   -8     2     0     0     1    -1     2     1   -95    -1]
[   -2   -12     0     0     1    -1     1    -1    -2    -1]
[    4    -4    -6     5     0     0    -2     0     1    -4]
[   -6     1    -1     1     1    -1     1    -1    -3     1]
[    1     0     0    -3     2    -2     0    -2     1     0]
[   -1     1     0     0     1    -1     4    -1     1    -1]
[   14     1    -5     4    -1     0     2     4     1     1]
[   -2    -1     0     4    -3     1    -5     0    -2    -1]
[   -9    -1    -1     3     2     1    -1     1    -2     1]
[   -1     2    -7     1     0     2     3 -1955   -22    -1]

sage: F = FP_LLL(A)
sage: F.fast_early_red()
sage: L = F._sage_(); L
[   1    0    0   -3    2   -2    0   -2    1    0]
[  -1    1    0    0    1   -1    4   -1    1   -1]
[  -2    0    0    1    0   -2   -1   -3    0   -2]
[  -2   -2    0   -1    3    0   -2    0    2    0]
[   1    1    1    2    3   -2   -2    0    3    1]
[  -4    1   -1    0    1    1    2    2   -3    3]
[   1   -3   -7    2    3   -1    0    0   -1   -1]
[   1   -9    1    3    1   -3    1   -1   -1    0]
[   8    5   19    3   27    6   -3    8  -25  -22]
[ 172  -25   57  248  261  793   76 -839  -41  376]

sage: L.is_LLL_reduced(eta=0.51,delta=0.99)
True
sage: L.hermite_form() == A.hermite_form()
True
```

heuristic (*precision=0*, *eta=0.51*, *delta=0.99*, *implementation=None*)

Perform LLL reduction using fpLLL's heuristic implementation.

INPUT:

- `precision` – (default: auto) internal precision
- `eta` – (default: `0.51`) LLL parameter η with $1/2 \le \eta < \sqrt{\delta}$
- `delta` – (default: `0.99`) LLL parameter δ with $1/4 < \delta \le 1$
- `implementation` – (default: `"mpfr"`) which floating point implementation to use, can be one of the following:
  - "double"
  - "dpe"
  - "mpfr"

OUTPUT:

Nothing is returned but the internal state modified.

EXAMPLE:

```
sage: from sage.libs.fplll.fplll import FP_LLL
sage: A = random_matrix(ZZ,10,10); A
[   -8      2      0      0      1     -1      2      1    -95     -1]
[   -2    -12      0      0      1     -1      1     -1     -2     -1]
[    4     -4     -6      5      0      0     -2      0      1     -4]
[   -6      1     -1      1      1     -1      1     -1     -3      1]
[    1      0      0     -3      2     -2      0     -2      1      0]
[   -1      1      0      0      1     -1      4     -1      1     -1]
[   14      1     -5      4     -1      0      2      4      1      1]
[   -2     -1      0      4     -3      1     -5      0     -2     -1]
[   -9     -1     -1      3      2      1     -1      1     -2      1]
[   -1      2     -7      1      0      2      3  -1955    -22     -1]

sage: F = FP_LLL(A)
sage: F.heuristic()
sage: L = F._sage_(); L
[   1    0    0   -3    2   -2    0   -2    1    0]
[  -1    1    0    0    1   -1    4   -1    1   -1]
[  -2    0    0    1    0   -2   -1   -3    0   -2]
[  -2   -2    0   -1    3    0   -2    0    2    0]
[   1    1    1    2    3   -2   -2    0    3    1]
[  -4    1   -1    0    1    1    2    2   -3    3]
[   1   -3   -7    2    3   -1    0    0   -1   -1]
[   1   -9    1    3    1   -3    1   -1   -1    0]
[   8    5   19    3   27    6   -3    8  -25  -22]
[ 172  -25   57  248  261  793   76 -839  -41  376]

sage: L.is_LLL_reduced(eta=0.51,delta=0.99)
True
sage: L.hermite_form() == A.hermite_form()
True
```

heuristic_early_red(*precision=0*, *eta=0.51*, *delta=0.99*, *implementation=None*)

Perform LLL reduction using fpLLL's heuristic implementation with early reduction.

This implementation inserts some early reduction steps inside the execution of the 'fast' LLL algorithm. This sometimes makes the entries of the basis smaller very quickly. It occurs in particular for lattice bases built from minimal polynomial or integer relation detection problems.

INPUT:

- `precision` – (default: auto) internal precision
- `eta` – (default: `0.51`) LLL parameter η with $1/2 \leq \eta < \sqrt{\delta}$
- `delta` – (default: `0.99`) LLL parameter δ with $1/4 < \delta \leq 1$
- `implementation` – (default: `"mpfr"`) which floating point implementation to use, can be one of the following:
  - `"double"`
  - `"dpe"`
  - `"mpfr"`

OUTPUT:

Nothing is returned but the internal state modified.

EXAMPLE:

```
sage: from sage.libs.fplll.fplll import FP_LLL
sage: A = random_matrix(ZZ,10,10); A
[   -8     2     0     0     1    -1     2     1   -95    -1]
[   -2   -12     0     0     1    -1     1    -1    -2    -1]
[    4    -4    -6     5     0     0    -2     0     1    -4]
[   -6     1    -1     1     1    -1     1    -1    -3     1]
[    1     0     0    -3     2    -2     0    -2     1     0]
[   -1     1     0     0     1    -1     4    -1     1    -1]
[   14     1    -5     4    -1     0     2     4     1     1]
[   -2    -1     0     4    -3     1    -5     0    -2    -1]
[   -9    -1    -1     3     2     1    -1     1    -2     1]
[   -1     2    -7     1     0     2     3 -1955   -22    -1]

sage: F = FP_LLL(A)
sage: F.heuristic_early_red()
sage: L = F._sage_(); L
[   1    0    0   -3    2   -2    0   -2    1    0]
[  -1    1    0    0    1   -1    4   -1    1   -1]
[  -2    0    0    1    0   -2   -1   -3    0   -2]
[  -2   -2    0   -1    3    0   -2    0    2    0]
[   1    1    1    2    3   -2   -2    0    3    1]
[  -4    1   -1    0    1    1    2    2   -3    3]
[   1   -3   -7    2    3   -1    0    0   -1   -1]
[   1   -9    1    3    1   -3    1   -1   -1    0]
[   8    5   19    3   27    6   -3    8  -25  -22]
[ 172  -25   57  248  261  793   76 -839  -41  376]

sage: L.is_LLL_reduced(eta=0.51,delta=0.99)
True
sage: L.hermite_form() == A.hermite_form()
True
```

proved(*precision=0*, *eta=0.51*, *delta=0.99*, *implementation=None*)

Perform LLL reduction using fpLLL's `proved` implementation. This implementation is the only provable correct floating point implementation in the free software world. Provability is only guaranteed if the `'mpfr'` implementation is chosen.

INPUT:

- `precision` – (default: auto) internal precision
- `eta` – (default: `0.51`) LLL parameter η with $1/2 \leq \eta < \sqrt{\delta}$
- `delta` – (default: `0.99`) LLL parameter δ with $1/4 < \delta \leq 1$
- `implementation` – (default: `"mpfr"`) which floating point implementation to use, can be one of the following:
  - `"double"`
  - `"dpe"`
  - `"mpfr"`

OUTPUT:

Nothing is returned but the internal state modified.

EXAMPLE:

```
sage: from sage.libs.fplll.fplll import FP_LLL
sage: A = random_matrix(ZZ,10,10); A
[   -8      2      0      0      1     -1      2      1    -95     -1]
[   -2    -12      0      0      1     -1      1     -1     -2     -1]
[    4     -4     -6      5      0      0     -2      0      1     -4]
[   -6      1     -1      1      1     -1      1     -1     -3      1]
[    1      0      0     -3      2     -2      0     -2      1      0]
[   -1      1      0      0      1     -1      4     -1      1     -1]
[   14      1     -5      4     -1      0      2      4      1      1]
[   -2     -1      0      4     -3      1     -5      0     -2     -1]
[   -9     -1     -1      3      2      1     -1      1     -2      1]
[   -1      2     -7      1      0      2      3  -1955    -22     -1]

sage: F = FP_LLL(A)
sage: F.proved()
sage: L = F._sage_(); L
[   1    0    0   -3    2   -2    0   -2    1    0]
[  -1    1    0    0    1   -1    4   -1    1   -1]
[  -2    0    0    1    0   -2   -1   -3    0   -2]
[  -2   -2    0   -1    3    0   -2    0    2    0]
[   1    1    1    2    3   -2   -2    0    3    1]
[  -4    1   -1    0    1    1    2    2   -3    3]
[   1   -3   -7    2    3   -1    0    0   -1   -1]
[   1   -9    1    3    1   -3    1   -1   -1    0]
[   8    5   19    3   27    6   -3    8  -25  -22]
[ 172  -25   57  248  261  793   76 -839  -41  376]

sage: L.is_LLL_reduced()
True
sage: L.hermite_form() == A.hermite_form()
True
```

shortest_vector(*method=None*)

Return a shortest vector.

INPUT:

- `method` - (default: `"proved"`) `"proved"` or `"fast"`

OUTPUT:

A shortest non-zero vector for this lattice.

EXAMPLE:

```
sage: from sage.libs.fplll.fplll import FP_LLL
sage: A = sage.crypto.gen_lattice(type='random', n=1, m=40, q=2^60, seed=42)
sage: F = FP_LLL(A)
sage: F.shortest_vector('proved')  == F.shortest_vector('fast')
True
```

wrapper(*precision=0, eta=0.51, delta=0.99*)

Perform LLL reduction using fpLLL's code{wrapper} implementation. This implementation invokes a sequence of floating point LLL computations such that

- the computation is reasonably fast (based on an heuristic model)
- the result is proven to be LLL reduced.

INPUT:

- `precision` – (default: auto) internal precision
- `eta` – (default: `0.51`) LLL parameter η with $1/2 \leq \eta < \sqrt{\delta}$
- `delta` – (default: `0.99`) LLL parameter δ with $1/4 < \delta \leq 1$

OUTPUT:

Nothing is returned but the internal state is modified.

EXAMPLE:

```
sage: from sage.libs.fplll.fplll import FP_LLL
sage: A = random_matrix(ZZ,10,10); A
[   -8     2     0     0     1    -1     2     1   -95    -1]
[   -2   -12     0     0     1    -1     1    -1    -2    -1]
[    4    -4    -6     5     0     0    -2     0     1    -4]
[   -6     1    -1     1     1    -1     1    -1    -3     1]
[    1     0     0    -3     2    -2     0    -2     1     0]
[   -1     1     0     0     1    -1     4    -1     1    -1]
[   14     1    -5     4    -1     0     2     4     1     1]
[   -2    -1     0     4    -3     1    -5     0    -2    -1]
[   -9    -1    -1     3     2     1    -1     1    -2     1]
[   -1     2    -7     1     0     2     3 -1955   -22    -1]

sage: F = FP_LLL(A)
sage: F.wrapper()
sage: L = F._sage_(); L
[   1    0    0   -3    2   -2    0   -2    1    0]
[  -1    1    0    0    1   -1    4   -1    1   -1]
[  -2    0    0    1    0   -2   -1   -3    0   -2]
[  -2   -2    0   -1    3    0   -2    0    2    0]
[   1    1    1    2    3   -2   -2    0    3    1]
[  -4    1   -1    0    1    1    2    2   -3    3]
[   1   -3   -7    2    3   -1    0    0   -1   -1]
[   1   -9    1    3    1   -3    1   -1   -1    0]
[   8    5   19    3   27    6   -3    8  -25  -22]
[ 172  -25   57  248  261  793   76 -839  -41  376]
sage: L.is_LLL_reduced()
True
sage: L.hermite_form() == A.hermite_form()
True
```

`sage.libs.fplll.fplll.`**`gen_ajtai`**(*d*, *alpha*)

Return Ajtai-like $(d \times d)$-matrix of floating point parameter α. The matrix is lower-triangular, B_{imi} is $2^{(d-i+1)^\alpha}$ and $B_{i,j}$ is $B_{j,j}/2$ for $j < i$.

INPUT:

- `d` – dimension
- `alpha` – the α above

OUTPUT:

An integer lattice.

EXAMPLE:

```
sage: from sage.libs.fplll.fplll import gen_ajtai
sage: A = gen_ajtai(10, 0.7); A # random output
[117   0   0   0   0   0   0   0   0   0]
[ 11  55   0   0   0   0   0   0   0   0]
[-47  21 104   0   0   0   0   0   0   0]
```

```
[ -3 -22 -16  95   0   0   0   0   0   0]
[ -8 -21  -3 -28  55   0   0   0   0   0]
[-33 -15 -30  37   8  52   0   0   0   0]
[-35  21  41 -31 -23  10  21   0   0   0]
[ -9  20 -34 -23 -18 -13  -9  63   0   0]
[-11  14 -38 -16 -26 -23  -3  11   9   0]
[ 15  21  35  37  12   6  -2  10   1  17]

sage: L = A.LLL(); L # random output
[  4   7  -3  21 -14 -17  -1  -1  -8  17]
[-20   0  -6   6 -11  -4 -19  10   1  17]
[-22  -1   8 -21  18 -29   3  11   9   0]
[ 31   8  20   2 -12  -4 -27 -22 -18   0]
[ -2   6  -4   7  -8 -10   6  52  -9   0]
[  3  -7  35 -12 -29  23   3  11   9   0]
[-16  -6 -16  37   2  11  -1  -9   7 -34]
[ 11  55   0   0   0   0   0   0   0   0]
[ 11  14  38  16  26  23   3  11   9   0]
[ 13 -28  -1   7 -11  11 -12   3  54   0]
sage: L.is_LLL_reduced()
True
sage: L.hermite_form() == A.hermite_form()
True
```

sage.libs.fplll.fplll.**gen_intrel**(d, b)

Return a $(d + 1 \times d)$-dimensional knapsack-type random lattice, where the x_i's are random b bits integers.

INPUT:

- d – dimension
- b – bitsize of entries

OUTPUT:

An integer lattice.

EXAMPLE:

```
sage: from sage.libs.fplll.fplll import gen_intrel
sage: A = gen_intrel(10,10); A
[116   1   0   0   0   0   0   0   0   0   0]
[331   0   1   0   0   0   0   0   0   0   0]
[303   0   0   1   0   0   0   0   0   0   0]
[963   0   0   0   1   0   0   0   0   0   0]
[456   0   0   0   0   1   0   0   0   0   0]
[225   0   0   0   0   0   1   0   0   0   0]
[827   0   0   0   0   0   0   1   0   0   0]
[381   0   0   0   0   0   0   0   1   0   0]
[ 99   0   0   0   0   0   0   0   0   1   0]
[649   0   0   0   0   0   0   0   0   0   1]

sage: L = A.LLL(); L
[ 1  1  1  0  0  0  0 -1  1  0  0]
[ 1  0  1  0  0 -1  1  0  0 -1  0]
[ 0  0  1  1  0 -1  0 -1  0  0  1]
[ 0 -1  0 -1 -1  1  0  1  0  1  0]
[-1 -1  0 -1  0 -1  1  0  0  0  1]
[ 0  1 -1  0  0 -1  1  1 -1  0  0]
[ 0  0  0  0 -1  1  1  0  1 -1  0]
[ 1 -1 -1  0  0 -1 -1  0  1  1  1]
```

```
[-1  0  0 -1 -1  0 -1  1  2 -1  0]
[-1 -1  0  0  1  0  2  0  0  0 -2]
sage: L.is_LLL_reduced()
True
sage: L.hermite_form() == A.hermite_form()
True
```

sage.libs.fplll.fplll.**gen_ntrulike**(d, b, q)

Generate a NTRU-like lattice of dimension $(2d \times 2d)$, with the coefficients h_i chosen as random b bits integers and parameter q:

```
[[ 1 0 ... 0 h0      h1 ... h_{d-1} ]
 [ 0 1 ... 0 h1      h2 ... h0      ]
 [ ................................ ]
 [ 0 0 ... 1 h_{d-1} h0 ... h_{d-1} ]
 [ 0 0 ... 0 q       0  ...  0      ]
 [ 0 0 ... 0 0       q  ...  0      ]
 [ ................................ ]
 [ 0 0 ... 0 0       0  ...  q      ]]
```

INPUT:

•d – dimension

•b – bitsize of entries

•q – the q above

OUTPUT:

An integer lattice.

EXAMPLE:

```
sage: from sage.libs.fplll.fplll import gen_ntrulike
sage: A = gen_ntrulike(5,10,12); A
[  1   0   0   0   0 320 351 920 714  66]
[  0   1   0   0   0 351 920 714  66 320]
[  0   0   1   0   0 920 714  66 320 351]
[  0   0   0   1   0 714  66 320 351 920]
[  0   0   0   0   1  66 320 351 920 714]
[  0   0   0   0   0  12   0   0   0   0]
[  0   0   0   0   0   0  12   0   0   0]
[  0   0   0   0   0   0   0  12   0   0]
[  0   0   0   0   0   0   0   0  12   0]
[  0   0   0   0   0   0   0   0   0  12]

sage: L = A.LLL(); L
[-1 -1  0  0  0  1  1 -2  0 -2]
[-1  0  0  0 -1 -2  1  1 -2  0]
[ 0 -1 -1  0  0  1 -2  0 -2  1]
[ 0  0  1  1  0  2  0  2 -1 -1]
[ 0  0  0  1  1  0  2 -1 -1  2]
[-2 -1 -2  1  1  1  0  1  1  0]
[-1 -2  1  1 -2  0  1  0  1  1]
[ 2 -1 -1  2  1 -1  0 -1  0 -1]
[-1 -1  2  1  2 -1 -1  0 -1  0]
[ 1 -2 -1 -2  1  0  1  1  0  1]
sage: L.is_LLL_reduced()
True
```

```
sage: L.hermite_form() == A.hermite_form()
True
```

sage.libs.fplll.fplll.**gen_ntrulike2**(*d*, *b*, *q*)

Like gen_ntrulike() but with the q vectors coming first.

INPUT:

•d – dimension

•b – bitsize of entries

•q – see gen_ntrulike()

OUTPUT:

An integer lattice.

EXAMPLE:

```
sage: from sage.libs.fplll.fplll import gen_ntrulike2
sage: A = gen_ntrulike2(5,10,12); A
[ 12   0   0   0   0   0   0   0   0   0]
[  0  12   0   0   0   0   0   0   0   0]
[  0   0  12   0   0   0   0   0   0   0]
[  0   0   0  12   0   0   0   0   0   0]
[  0   0   0   0  12   0   0   0   0   0]
[902 947 306  40 908   1   0   0   0   0]
[947 306  40 908 902   0   1   0   0   0]
[306  40 908 902 947   0   0   1   0   0]
[ 40 908 902 947 306   0   0   0   1   0]
[908 902 947 306  40   0   0   0   0   1]

sage: L = A.LLL(); L
[ 1  0  0  2 -3 -2  1  1  0  0]
[-1  0 -2  1  2  2  1 -2 -1  0]
[ 0  2 -1 -2  1  0 -2 -1  2  1]
[ 0  3  0  1  3  1  0 -1  1  0]
[ 2 -1  0 -2  1  1 -2 -1  0  2]
[ 0 -1  0 -1 -1  1  4 -1 -1  0]
[ 2  1  1  1 -1 -3 -2 -1 -1 -1]
[-1  0 -1  0 -1  4 -1 -1  0  1]
[ 0  1 -2  1  1 -1  0  1 -3 -2]
[-2  1  1  0  1 -3 -2 -1  0  1]
sage: L.is_LLL_reduced()
True
sage: L.hermite_form() == A.hermite_form()
True
```

sage.libs.fplll.fplll.**gen_simdioph**(*d*, *b*, *b2*)

Return a d-dimensional simultaneous diophantine approximation random lattice, where the d x_i's are random b bits integers.

INPUT:

•d – dimension

•b – bitsize of entries

•b2 – bitsize of entries

OUTPUT:

An integer lattice.

EXAMPLE:

```
sage: from sage.libs.fplll.fplll import gen_simdioph
sage: A = gen_simdioph(10,10,3); A
[   8  395  975  566  213  694  254  629  303  597]
[   0 1024    0    0    0    0    0    0    0    0]
[   0    0 1024    0    0    0    0    0    0    0]
[   0    0    0 1024    0    0    0    0    0    0]
[   0    0    0    0 1024    0    0    0    0    0]
[   0    0    0    0    0 1024    0    0    0    0]
[   0    0    0    0    0    0 1024    0    0    0]
[   0    0    0    0    0    0    0 1024    0    0]
[   0    0    0    0    0    0    0    0 1024    0]
[   0    0    0    0    0    0    0    0    0 1024]

sage: L = A.LLL(); L
[ 192  264 -152  272   -8  272  -48 -264  104   -8]
[-128 -176 -240  160 -336  160   32  176  272 -336]
[ -24 -161  147  350  385  -34  262  161  115  257]
[ 520   75 -113  -74 -491   54  126  -75  239 -107]
[-376 -133  255   22  229  150  350  133   95 -411]
[-168 -103    5  402 -377 -238 -214  103 -219 -249]
[-352   28  108 -328 -156  184   88  -28  -20  356]
[ 120 -219  289  298  123  170 -286  219  449 -261]
[ 160 -292   44   56  164  568  -40  292  -84 -348]
[-192  760  152 -272    8 -272   48  264 -104    8]
sage: L.is_LLL_reduced()
True
sage: L.hermite_form() == A.hermite_form()
True
```

sage.libs.fplll.fplll.**gen_uniform**(*nr*, *nc*, *b*)

Return a $(nr \times nc)$ matrix where the entries are random `b` bits integers.

INPUT:

- `nr` – row dimension
- `nc` – column dimension
- `b` – bitsize of entries

OUTPUT:

An integer lattice.

EXAMPLE:

```
sage: from sage.libs.fplll.fplll import gen_uniform
sage: A = gen_uniform(10,10,12); A
[ 980 3534  533 3303 2491 2960 1475 3998  105  162]
[1766 3683 2782  668 2356 2149 1887 2327  976 1151]
[1573  438 1480  887 1490  634 3820 3379 4074 2669]
[ 215 2054 2388 3214 2459  250 2921 1395 3626  434]
[ 638 4011 3626 1864  633 1072 3651 2339 2926 1004]
[3731  439 1087 1088 2627 3446 2669 1419  563 2079]
[1868 3196 3712 4016 1451 2589 3327  712  647 1057]
[2068 2761 3479 2552  197 1258 1544 1116 3090 3667]
[1394  529 1683 1781 1779 3032   80 2712  639 3047]
[3695 3888 3139  851 2111 3375  208 3766 3925 1465]
```

```
sage: L = A.LLL(); L
[  200 -1144  -365   755  1404  -218  -937   321  -718   790]
[  623   813   873  -595  -422   604  -207  1265 -1418  1360]
[ -928  -816   479  1951  -319 -1295   827   333  1232   643]
[-1802 -1904  -952   425  -141   697   300  1608  -501  -767]
[ -572 -2010  -734   358 -1981  1101  -870    64   381  1106]
[  853  -223   767  1382  -529  -780  -500  1507 -2455 -1190]
[-1016 -1755  1297 -2210  -276  -114   712   -63   370   222]
[ -430  1471   339  -513  1361  2715  2076  -646 -1406   -60]
[-3390   748    62   775   935  1697  -306  -618    88  -452]
[  713 -1115  1887  -563   733  2443   816   972   876 -2074]
sage: L.is_LLL_reduced()
True
sage: L.hermite_form() == A.hermite_form()
True
```

CHAPTER

FIVE

THE ELLIPTIC CURVE METHOD FOR INTEGER FACTORIZATION (ECM)

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:

- Robert L Miller (2008-01-21): library interface (clone of ecmfactor.c)
- Jeroen Demeyer (2012-03-29): signal handling, documentation

EXAMPLES:

```
sage: from sage.libs.libecm import ecmfactor
sage: result = ecmfactor(999, 0.00)
sage: result in [(True, 27), (True, 37), (True, 999)]
True
sage: result = ecmfactor(999, 0.00, verbose=True)
Performing one curve with B1=0
Found factor in step 1: ...
sage: result in [(True, 27), (True, 37), (True, 999)]
True
```

sage.libs.libecm.**ecmfactor**(*number*, *B1*, *verbose=False*)

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 * 79638304766856507377778616296087448490695649
sage: ecmfactor(N, 2e5)
(True, 2349023)
```

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 * 1489459109360039866456940197095433721664951999121
sage: ecmfactor(N, 1e3)  # random
(True, 514447)
```

We can ask for verbose output:

```
sage: N = 12^97 - 1
sage: factor(N)
11 * 43570062353753446053455610056679740005056966111842089407838902783209959981593077811330507
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)
```

TESTS:

Check that `ecmfactor` can be interrupted (factoring a large prime number):

```
sage: alarm(0.5); ecmfactor(2^521-1, 1e7)
Traceback (most recent call last):
...
AlarmInterrupt
```

Some special cases:

```
sage: ecmfactor(1, 100)
(True, 1)
sage: ecmfactor(0, 100)
Traceback (most recent call last):
...
ValueError: Input number (0) must be positive
```

CHAPTER

SIX

RUBINSTEIN'S LCALC LIBRARY

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
```

`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)
[6.64845334472..., 9.83144443288..., 11.9588456260...]

sage: L=Lfunction_from_character(chi, type="double")
sage: L.find_zeros(1,15,.1)
[6.64845334472..., 9.83144443288..., 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 fomula 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:

```
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:

```
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.hardy_z_function(0)
0.231750947504...
sage: L.hardy_z_function(.5).imag().abs() < 1.0e-16
True
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)
0.7939675904771...
sage: L.hardy_z_function(.5).imag().abs() < 1.0e-16
True
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() < 1.0e-16
True
```

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:

```
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)
0.231750947504... + 5.75329642226...e-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)
0.274633355856... + 6.59869267328...e-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.

$$\Lambda(s) = \omega Q^s \overline{\Lambda(1-\bar{s})}$$

where

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

See (23) in http://arxiv.org/abs/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 κ_j in the functional equation
- `gamma` - List of the values of γ_j in the functional equation
- `pole` - List of the poles of L-function
- `residue` - List of the residues of the L-function

NOTES: 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.

$$\Lambda(s) = \omega Q^s \overline{\Lambda(1-\bar{s})}$$

where

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

See (23) in http://arxiv.org/abs/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 κ_j in the functional equation
- `gamma` - List of the values of γ_j in the functional equation
- `pole` - List of the poles of L-function
- `residue` - List of the residues of the L-function

NOTES: 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.

$$\Lambda(s) = \omega Q^s \overline{\Lambda(1-\bar{s})}$$

where

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

See (23) in http://arxiv.org/abs/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 κ_j in the functional equation
- `gamma` - List of the values of γ_j in the functional equation
- `pole` - List of the poles of L-function
- `residue` - List of the residues of the L-function

NOTES:

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:

```
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:

```
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...
```

CHAPTER

SEVEN

AN INTERFACE TO ANDERS BUCH'S LITTLEWOOD-RICHARDSON CALCULATOR LRCALC

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://math.rutgers.edu/~asbuch/lrcalc/.

The following describes the Sage interface to this library.

EXAMPLES:

```
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])
{[3, 3]: 1, [4, 2]: 1, [3, 1, 1, 1]: 1, [4, 1, 1]: 1, [2, 2, 2]: 1, [3, 2, 1]: 2, [2, 2, 1, 1]: 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)
{[3, 3]: 1, [4, 2]: 1, [3, 2, 1]: 2, [2, 2, 2]: 1, [4, 1, 1]: 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])
{[3]: 1, [1, 1, 1]: 1, [2, 1]: 2}
```

Compute the coproduct of a Schur function:

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

Multiply two Schubert polynomials:

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

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

```
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],
```

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:

```
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:

```
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
     22222    <--- 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} * 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`**`(`*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:

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

```
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:

```
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
```

`sage.libs.lrcalc.lrcalc.`**`lrskew`**(*outer*, *inner*, *weight=None*, *maxrows=0*)

Return the skew LR tableaux of shape `outer / inner`.

INPUT:

- `outer` – a partition.
- `inner` – a partition.
- `weight` – a partition (optional).
- `maxrows` – an integer (optional).

OUTPUT: a list of `SkewTableau`x. This will change to an iterator over such skew tableaux once Cython will support the ``yield`` statement. Specifying a third entry $maxrows$ restricts the alphabet to $\{1, 2, \dots, maxrows\}$. Specifying $weight$ returns only those tableaux of given content/weight.

EXAMPLES:

```
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: lrskew([3,2,1],[2], weight=[3,1])
[[[None, None, 1], [1, 1], [2]]]
```

`sage.libs.lrcalc.lrcalc.`**`mult`**(*part1*, *part2*, *maxrows=None*, *level=None*)

Compute a product of two Schur functions.

Return the product of the Schur functions indexed by the partitions `part1` and `part2`.

INPUT:

- `part1` – a partition.
- `part2` – a partition.
- `maxrows` – an integer or None.
- `level` – an integer or None.

If `maxrows` is specified, then only partitions with at most this number of rows is included in the result.

If both `maxrows` and `level` are specified, then the function calculates the fusion product for $sl(\text{maxrows})$ of the given level.

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)]
sage: sorted(mult([2,1],[2,1]).items())
[([2, 2, 1, 1], 1), ([2, 2, 2], 1), ([3, 1, 1, 1], 1), ([3, 2, 1], 2), ([3, 3], 1), ([4, 1, 1],
sage: sorted(mult([2,1],[2,1],maxrows=2).items())
[([3, 3], 1), ([4, 2], 1)]
sage: mult([2,1],[3,2,1],3)
{[3, 3, 3]: 1, [5, 2, 2]: 1, [5, 3, 1]: 1, [4, 4, 1]: 1, [4, 3, 2]: 2}
sage: mult([2,1],[2,1],3,3)
{[3, 3]: 1, [3, 2, 1]: 2, [2, 2, 2]: 1, [4, 1, 1]: 1}
sage: mult([2,1],[2,1],None,3)
Traceback (most recent call last):
...
ValueError: maxrows needs to be specified if you specify the level
```

sage.libs.lrcalc.lrcalc.**mult_schubert**(*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 `w1` and `w2`.

INPUT:

- `w1` – a permutation.
- `w2` – a permutation.
- `rank` – an integer.

If `rank` is non-zero, then only permutations from the symmetric group $S(\mathrm{rank})$ are included in the result.

EXAMPLES:

```
sage: from sage.libs.lrcalc.lrcalc import mult_schubert
sage: result = mult_schubert([3, 1, 5, 2, 4], [3, 5, 2, 1, 4])
sage: sorted(result.items())
[([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 `outer / inner`, consisting of boxes in the partition `outer` that are not in `inner`.

INPUT:

- `outer` – a partition.
- `inner` – a partition.
- `maxrows` – an integer or None.

If `maxrows` is specified, then only partitions with at most this number of rows are included in the result.

EXAMPLES:

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

```
sage: from sage.libs.lrcalc.lrcalc import test_iterable_to_vector
sage: x = test_iterable_to_vector([3,2,1]); x
[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 `outer/inner` as an `lrcalc` `skewtab`, 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
```

CHAPTER

EIGHT

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.

CHAPTER

NINE

CREMONA'S MWRANK C++ LIBRARY

`sage.libs.mwrank.all.`**`mwrank_initprimes`**(*filename*, *verb=False*)

mwrank_initprimes(filename, verb=False):

INPUT:

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

EXAMPLES:

```
sage: file = tmp_filename()
sage: open(file,'w').write(' '.join([str(p) for p in prime_range(10^6)]))
sage: mwrank_initprimes(file, verb=False)
```

CHAPTER

TEN

CYTHON INTERFACE TO CREMONA'S `ECLIB` LIBRARY (ALSO KNOWN AS `MWRANK`)

Cython interface to Cremona's `eclib` library (also known as `mwrank`)

EXAMPLES:

```
sage: from sage.libs.mwrank.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

sage: t= _mw(c)
sage: t.search(10)
sage: t
[[1:2:1]]
```

`sage.libs.mwrank.mwrank.`**`get_precision`**`()`

Returns the working floating point precision of mwrank.

OUTPUT:

(int) The current precision in decimal digits.

EXAMPLE:

```
sage: from sage.libs.mwrank.mwrank import get_precision
sage: get_precision()
50
```

`sage.libs.mwrank.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:

```
sage: file = os.path.join(SAGE_TMP, 'PRIMES')
sage: open(file,'w').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.mwrank.mwrank.**set_precision**(*n*)

Sets the working floating point precision of mwrank.

INPUT:

- `n` (int) – a positive integer: the number of decimal digits.

OUTPUT:

None.

EXAMPLE:

```
sage: from sage.libs.mwrank.mwrank import set_precision
sage: set_precision(50)
```

CHAPTER

ELEVEN

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`.

`sage.libs.mwrank.interface.`**`get_precision`**`()`

Return the global NTL real number precision.

See also `set_precision()`.

Warning: The internal precision is binary. This function multiplies the binary precision by 0.3 ($= \log_2(10)$ approximately) and truncates.

OUTPUT:

(int) The current decimal precision.

EXAMPLES:

```
sage: mwrank_get_precision()
50
```

class `sage.libs.mwrank.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, and 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 $y^2 + y = x^3 + x^2 - 2x$:

```
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:

```
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 two_descent() call provably correctly computed the rank. If `two_descent()` hasn't been called, then it is first called by `certain()` using the default parameters.

The result is `True` if and only if the results of the methods `rank()` and `rank_bound()` are equal.

EXAMPLES:

A 2-descent does not determine $E(\mathbf{Q})$ 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)
...
sage: E.certain()
False
sage: E.rank()
0
```

The previous value is only a lower bound; the upper bound is greater:

```
sage: E.rank_bound()
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()
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`(*verbose=False*)

Returns the isogeny class of this mwrank elliptic curve.

EXAMPLES:

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

`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()
0
sage: E.certain()
True

sage: E = mwrank_EllipticCurve([0, -1, 1, -929, -10595])
sage: E.rank()
0
sage: E.certain()
False
```

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()
0.0511114082399688
```

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:

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

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:

```
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:

```
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`:

```
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:

```
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:

```
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:

```
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-14
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 = 0.051111408239968840235
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
Selmer  rank contribution from B=im(eps) = 1
Sha     rank contribution from B=im(eps) = 0
Mordell rank contribution from A=ker(eps) = 0
Selmer  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 0.051111408239968840235
Processing points found during 2-descent...done:
  now regulator = 0.051111408239968840235
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:

```
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.

TESTS:

See trac ticket #7992:

```
sage: EllipticCurve([0, prod(prime_range(10))]).mwrank_curve().two_descent()
Basic pair: I=0, J=-5670
disc=-32148900
2-adic index bound = 2
2-adic index = 2
Two (I,J) pairs
Looking for quartics with I = 0, J = -5670
Looking for Type 3 quartics:
Trying positive a from 1 up to 5 (square a first...)
Trying positive a from 1 up to 5 (...then non-square a)
(2,0,-12,19,-6)      --nontrivial...(x:y:z) = (2 : 4 : 1)
Point = [-2488:-4997:512]
    height = 6.46767239...
Rank of B=im(eps) increases to 1
Trying negative a from -1 down to -3
Finished looking for Type 3 quartics.
Looking for quartics with I = 0, J = -362880
Looking for Type 3 quartics:
Trying positive a from 1 up to 20 (square a first...)
Trying positive a from 1 up to 20 (...then non-square a)
Trying negative a from -1 down to -13
Finished looking for Type 3 quartics.
Mordell rank contribution from B=im(eps) = 1
Selmer  rank contribution from B=im(eps) = 1
Sha     rank contribution from B=im(eps) = 0
Mordell rank contribution from A=ker(eps) = 0
```

```
Selmer  rank contribution from A=ker(eps) = 0
Sha     rank contribution from A=ker(eps) = 0
sage: EllipticCurve([0, prod(prime_range(100))]).mwrank_curve().two_descent()
Traceback (most recent call last):
...
RuntimeError: Aborted
```

Calling this method twice does not cause a segmentation fault (see trac ticket #10665):

```
sage: E = EllipticCurve([1, 1, 0, 0, 528])
sage: E.two_descent(verbose=False)
True
sage: E.two_descent(verbose=False)
True
```

class `sage.libs.mwrank.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 $\mathbf{Z}$-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).

EXAMPLE:

```
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:

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

```
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
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)
sage: EQ
Subgroup of Mordell-Weil group: [[1:-1:1], [-2:3:1], [-14:25:8]]
```

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

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E, verbose=False, pp=1)
sage: EQ.search(1); EQ # generators only
Subgroup of Mordell-Weil group: [[1:-1:1], [-2:3:1], [-14:25:8]]
sage: EQ = mwrank_MordellWeil(E, verbose=False, pp=0)
sage: EQ.search(1); EQ # all points found
Subgroup of Mordell-Weil group: [[-3:0:1], [-2:3:1], [-14:25:8], [-1:3:1], [0:2:1], [2:13:8], [1
```

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 `mwrank_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 `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]])
P1 = [1:-1:1]         is generator number 1
P2 = [-2:3:1]         is generator number 2
P3 = [-14:25:8]       is generator number 3

sage: EQ.points()
[[1, -1, 1], [-2, 3, 1], [-14, 25, 8]]
```

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, 2969715140223272],
P1 = [1547:-2967:343]         is generator number 1
...
Gained index 5, new generators = [ [-2:3:1] [-14:25:8] [1:-1:1] ]

sage: EQ.points()
[[-2, 3, 1], [-14, 25, 8], [1, -1, 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],
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, -
sage: EQ.regulator()
375.42919921875
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 =  93.857300720636393209
(False, 2, '[ ]')
sage: EQ.points()
[[-2, 3, 1], [2707496766203306, 864581029138191, 2969715140223272], [-13422227300, -4932283
sage: EQ.regulator()
93.8572998046875
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 =  10.4285889689595992455
(False, 3, '[ ]')
sage: EQ.points()
[[-2, 3, 1], [-14, 25, 8], [-13422227300, -49322830557, 12167000000]]
sage: EQ.regulator()
10.4285888671875
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 =  0.417143558758383969818
(False, 5, '[ ]')
sage: EQ.points()
[[-2, 3, 1], [-14, 25, 8], [1, -1, 1]]
sage: EQ.regulator()
0.4171435534954071
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, '[ ]')
```

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:

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

A rank 3 example:

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.rank()
0
sage: EQ.regulator()
1.0
```

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

```
sage: EQ
Subgroup of Mordell-Weil group: []
```

Now we do a very small search:

```
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()
0.4171435534954071
```

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

regulator()

Return the regulator of the points in this subgroup of the Mordell-Weil group.

Note: eclib can compute the regulator to arbitrary precision, but the interface currently returns the output as a float.

OUTPUT:

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

EXAMPLES:

```
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 :meth: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 decimal 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:

```
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):

```
sage: EQ.process([[1547, -2967, 343], [2707496766203306, 864581029138191, 2969715140223272],
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
Subgroup of Mordell-Weil group: [[1547:-2967:343], [2707496766203306:864581029138191:2969715
sage: EQ.regulator()
375.42919921875
```

Now we saturate at $p = 2$, and gain index 2:

```
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 =  93.857300720636393209
(False, 2, '[ ]')
sage: EQ
Subgroup of Mordell-Weil group: [[-2:3:1], [2707496766203306:864581029138191:296971514022327
sage: EQ.regulator()
93.8572998046875
```

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

```
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 =  10.428588968959599245
(False, 3, '[ ]')
sage: EQ
Subgroup of Mordell-Weil group: [[-2:3:1], [-14:25:8], [-13422227300:-49322830557:121670000
sage: EQ.regulator()
10.4285888671875
```

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

```
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 =  0.41714355875838396981
(False, 5, '[ ]')
sage: EQ
Subgroup of Mordell-Weil group: [[-2:3:1], [-14:25:8], [1:-1:1]]
sage: EQ.regulator()
0.4171435534954071
```

Finally we finish the saturation. The output here shows that the points are now provably saturated at all primes:

```
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:

```
sage: E = mwrank_EllipticCurve([0,0,1,-7,6])
sage: EQ = mwrank_MordellWeil(E)
sage: EQ.process([[1547, -2967, 343], [2707496766203306, 864581029138191, 2969715140223272],
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()
0.4171435534954071
```

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

```
sage: EQ.saturate()
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`.

- `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:

```
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:

```
sage: E = mwrank_EllipticCurve([0, -1, 0, -18392, -1186248]) #1056g4
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]]
```

`sage.libs.mwrank.interface.`**`set_precision`**`(n)`

Set the global NTL real number precision. This has a massive effect on the speed of mwrank calculations. The default (used if this function is not called) is `n=50`, but it might have to be increased if a computation fails. See also `get_precision()`.

INPUT:

- `n` (long) – real precision used for floating point computations in the library, in decimal digits.

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

EXAMPLES:

```
sage: mwrank_set_precision(20)
```

CHAPTER

TWELVE

PARI C-LIBRARY INTERFACE

PARI C-library interface

AUTHORS:

- William Stein (2006-03-01): updated to work with PARI 2.2.12-beta
- William Stein (2006-03-06): added newtonpoly
- Justin Walker: contributed some of the function definitions
- Gonzalo Tornaria: improvements to conversions; much better error handling.
- Robert Bradshaw, Jeroen Demeyer, William Stein (2010-08-15): Upgrade to PARI 2.4.3 (#9343)
- Jeroen Demeyer (2011-11-12): rewrite various conversion routines (#11611, #11854, #11952)
- Peter Bruin (2013-11-17): split off this file from gen.pyx (#15185)

EXAMPLES:

```
sage: pari('5! + 10/x')
(120*x + 10)/x
sage: pari('intnum(x=0,13,sin(x)+sin(x^2) + x)')
85.1885681951527
sage: f = pari('x^3-1')
sage: v = f.factor(); v
[x - 1, 1; x^2 + x + 1, 1]
sage: v[0]   # indexing is 0-based unlike in GP.
[x - 1, x^2 + x + 1]~
sage: v[1]
[1, 1]~
```

Arithmetic obeys the usual coercion rules:

```
sage: type(pari(1) + 1)
<type 'sage.libs.pari.gen.gen'>
sage: type(1 + pari(1))
<type 'sage.libs.pari.gen.gen'>
```

GUIDE TO REAL PRECISION AND THE PARI LIBRARY

The default real precision in communicating with the Pari library is the same as the default Sage real precision, which is 53 bits. Inexact Pari objects are therefore printed by default to 15 decimal digits (even if they are actually more precise).

Default precision example (53 bits, 15 significant decimals):

```
sage: a = pari(1.23); a
1.23000000000000
sage: a.sin()
0.942488801931698
```

Example with custom precision of 200 bits (60 significant decimals):

```
sage: R = RealField(200)
sage: a = pari(R(1.23)); a   # only 15 significant digits printed
1.23000000000000
sage: R(a)         # but the number is known to precision of 200 bits
1.2300000000000000000000000000000000000000000000000000000000
sage: a.sin()      # only 15 significant digits printed
0.942488801931698
sage: R(a.sin())   # but the number is known to precision of 200 bits
0.94248880193169751002382356538924454146128740562765030213504
```

It is possible to change the number of printed decimals:

```
sage: R = RealField(200)    # 200 bits of precision in computations
sage: old_prec = pari.set_real_precision(60)  # 60 decimals printed
sage: a = pari(R(1.23)); a
1.23000000000000000000000000000000000000000000000000000000000
sage: a.sin()
0.942488801931697510023823565389244541461287405627650302135038
sage: pari.set_real_precision(old_prec)  # restore the default printing behavior
60
```

Unless otherwise indicated in the docstring, most Pari functions that return inexact objects use the precision of their arguments to decide the precision of the computation. However, if some of these arguments happen to be exact numbers (integers, rationals, etc.), an optional parameter indicates the precision (in bits) to which these arguments should be converted before the computation. If this precision parameter is missing, the default precision of 53 bits is used. The following first converts 2 into a real with 53-bit precision:

```
sage: R = RealField()
sage: R(pari(2).sin())
0.909297426825682
```

We can ask for a better precision using the optional parameter:

```
sage: R = RealField(150)
sage: R(pari(2).sin(precision=150))
0.90929742682568169539601986591174484270225497
```

Warning regarding conversions Sage - Pari - Sage: Some care must be taken when juggling inexact types back and forth between Sage and Pari. In theory, calling p=pari(s) creates a Pari object p with the same precision as s; in practice, the Pari library's precision is word-based, so it will go up to the next word. For example, a default 53-bit Sage real s will be bumped up to 64 bits by adding bogus 11 bits. The function p.python() returns a Sage object with exactly the same precision as the Pari object p. So pari(s).python() is definitely not equal to s, since it has 64 bits of precision, including the bogus 11 bits. The correct way of avoiding this is to coerce pari(s).python() 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 in gen.pyx). For instance, if we want to use the Pari library to compute sqrt(pi) with a precision of 100 bits:

```
sage: R = RealField(100)
sage: s = R(pi); s
3.1415926535897932384626433833
sage: p = pari(s).sqrt()
```

```
sage: x = p.python(); 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
```

Elliptic curves and precision: If you are working with elliptic curves and want to compute with a precision other than the default 53 bits, you should use the precision parameter of ellinit():

```
sage: R = RealField(150)
sage: e = pari([0,0,0,-82,0]).ellinit(precision=150)
sage: eta1 = e.elleta()[0]
sage: R(eta1)
3.6054636014326520859158205642077267748102690
```

Number fields and precision: TODO

TESTS:

Check that output from PARI's print command is actually seen by Sage (ticket #9636):

```
sage: pari('print("test")')
test
```

class `sage.libs.pari.pari_instance.`**`PariInstance`**

Bases: `sage.structure.parent_base.ParentWithBase`

Initialize the PARI system.

INPUT:

- `size` – long, the number of bytes for the initial PARI stack (see note below)
- `maxprime` – unsigned long, upper limit on a precomputed prime number table (default: 500000)

Note: In Sage, the PARI stack is different than in GP or the PARI C library. In Sage, instead of the PARI stack holding the results of all computations, it *only* holds the results of an individual computation. Each time a new Python/PARI object is computed, it it copied to its own space in the Python heap, and the memory it occupied on the PARI stack is freed. Thus it is not necessary to make the stack very large. Also, unlike in PARI, if the stack does overflow, in most cases the PARI stack is automatically increased and the relevant step of the computation rerun.

This design obviously involves some performance penalties over the way PARI works, but it scales much better and is far more robust for large projects.

Note: If you do not want prime numbers, put `maxprime=2`, but be careful because many PARI functions require this table. If you get the error message "not enough precomputed primes", increase this parameter.

`allocatemem`(*s=0*, *silent=False*)

Change the PARI stack space to the given size (or double the current size if `s` is 0).

If $s = 0$ and insufficient memory is avaible to double, the PARI stack will be enlarged by a smaller amount. In any case, a `MemoryError` will be raised if the requested memory cannot be allocated.

The PARI stack is never automatically shrunk. You can use the command `pari.allocatemem(10^6)` to reset the size to 10^6, which is the default size at startup. Note that the results of computations using Sage's PARI interface are copied to the Python heap, so they take up no space in the PARI stack. The PARI stack is cleared after every computation.

It does no real harm to set this to a small value as the PARI stack will be automatically doubled when we run out of memory. However, it could make some calculations slower (since they have to be redone from the start after doubling the stack until the stack is big enough).

INPUT:

- `s` - an integer (default: 0). A non-zero argument should be the size in bytes of the new PARI stack. If s is zero, try to double the current stack size.

EXAMPLES:

```
sage: pari.allocatemem(10^7)
PARI stack size set to 10000000 bytes
sage: pari.allocatemem()  # Double the current size
PARI stack size set to 20000000 bytes
sage: pari.stacksize()
20000000
sage: pari.allocatemem(10^6)
PARI stack size set to 1000000 bytes
```

The following computation will automatically increase the PARI stack size:

```
sage: a = pari('2^100000000')
```

`a` is now a Python variable on the Python heap and does not take up any space on the PARI stack. The PARI stack is still large because of the computation of `a`:

```
sage: pari.stacksize()
16000000
sage: pari.allocatemem(10^6)
PARI stack size set to 1000000 bytes
sage: pari.stacksize()
1000000
```

TESTS:

Do the same without using the string interface and starting from a very small stack size:

```
sage: pari.allocatemem(1)
PARI stack size set to 1024 bytes
sage: a = pari(2)^100000000
sage: pari.stacksize()
16777216
```

complex(*re*, *im*)

Create a new complex number, initialized from re and im.

debugstack()

Print the internal PARI variables `top` (top of stack), `avma` (available memory address, think of this as the stack pointer), `bot` (bottom of stack).

EXAMPLE:

```
sage: pari.debugstack()  # random
top =  0x60b2c60
avma = 0x5875c38
bot =  0x57295e0
size = 1000000
```

default (*variable*, *value=None*)

double_to_gen (*x*)

euler (*precision=0*)

Return Euler's constant to the requested real precision (in bits).

EXAMPLES:

```
sage: pari.euler()
0.577215664901533
sage: pari.euler(precision=100).python()
0.577215664901532860606512090082...
```

factorial (*n*)

Return the factorial of the integer n as a PARI gen.

EXAMPLES:

```
sage: pari.factorial(0)
1
sage: pari.factorial(1)
1
sage: pari.factorial(5)
120
sage: pari.factorial(25)
15511210043330985984000000
```

get_debug_level ()

Set the debug PARI C library variable.

get_real_precision ()

Returns the current PARI default real precision.

This is used both for creation of new objects from strings and for printing. It is the number of digits *IN DECIMAL* in which real numbers are printed. It also determines the precision of objects created by parsing strings (e.g. pari('1.2')), which is *not* the normal way of creating new pari objects in Sage. It has *no* effect on the precision of computations within the pari library.

EXAMPLES:

```
sage: pari.get_real_precision()
15
```

get_series_precision ()

getrand ()

Returns PARI's current random number seed.

OUTPUT:

GEN of type t_VECSMALL

EXAMPLES:

```
sage: pari.setrand(50)
sage: a = pari.getrand(); a
Vecsmall([...])
sage: pari.setrand(a)
sage: a == pari.getrand()
True
```

init_primes (*_M*)

Recompute the primes table including at least all primes up to M (but possibly more).

EXAMPLES:

```
sage: pari.init_primes(200000)
```

We make sure that ticket #11741 has been fixed, and double check to make sure that diffptr has not been freed:

```
sage: pari.init_primes(2^62)
Traceback (most recent call last):
...
PariError: not enough memory                    # 64-bit
OverflowError: long int too large to convert   # 32-bit
sage: pari.init_primes(200000)
```

matrix(*m*, *n*, *entries=None*)
matrix(long m, long n, entries=None): Create and return the m x n PARI matrix with given list of entries.

new_with_bits_prec(*s*, *precision*)
pari.new_with_bits_prec(self, s, precision) creates s as a PARI gen with (at most) precision *bits* of precision.

nth_prime(*n*)

pari_version()

pi(*precision=0*)
Return the value of the constant pi to the requested real precision (in bits).

EXAMPLES:

```
sage: pari.pi()
3.14159265358979
sage: pari.pi(precision=100).python()
3.1415926535897932384626433832...
```

polcyclo(*n*, *v=-1*)
polcyclo(n, v=x): cyclotomic polynomial of degree n, in variable v.

EXAMPLES:

```
sage: pari.polcyclo(8)
x^4 + 1
sage: pari.polcyclo(7, 'z')
z^6 + z^5 + z^4 + z^3 + z^2 + z + 1
sage: pari.polcyclo(1)
x - 1
```

polcyclo_eval(*n*, *v*)
polcyclo_eval(n, v): value of the nth cyclotomic polynomial at value v.

EXAMPLES:

```
sage: pari.polcyclo_eval(8, 2)
17
sage: cyclotomic_polynomial(8)(2)
17
```

pollegendre(*n*, *v=-1*)
pollegendre(n, v=x): Legendre polynomial of degree n (n C-integer), in variable v.

EXAMPLES:

```
sage: pari.pollegendre(7)
429/16*x^7 - 693/16*x^5 + 315/16*x^3 - 35/16*x
sage: pari.pollegendre(7, 'z')
429/16*z^7 - 693/16*z^5 + 315/16*z^3 - 35/16*z
sage: pari.pollegendre(0)
1
```

polsubcyclo(*n*, *d*, *v=-1*)

polsubcyclo(n, d, v=x): return the pari list of polynomial(s) defining the sub-abelian extensions of degree d of the cyclotomic field $\mathbf{Q}(\zeta_n)$, where d divides $\phi(n)$.

EXAMPLES:

```
sage: pari.polsubcyclo(8, 4)
[x^4 + 1]
sage: pari.polsubcyclo(8, 2, 'z')
[z^2 - 2, z^2 + 1, z^2 + 2]
sage: pari.polsubcyclo(8, 1)
[x - 1]
sage: pari.polsubcyclo(8, 3)
[]
```

poltchebi(*n*, *v=-1*)

poltchebi(n, v=x): Chebyshev polynomial of the first kind of degree n, in variable v.

EXAMPLES:

```
sage: pari.poltchebi(7)
64*x^7 - 112*x^5 + 56*x^3 - 7*x
sage: pari.poltchebi(7, 'z')
64*z^7 - 112*z^5 + 56*z^3 - 7*z
sage: pari.poltchebi(0)
1
```

polzagier(*n*, *m*)

prime_list(*n*)

prime_list(n): returns list of the first n primes

To extend the table of primes use pari.init_primes(M).

INPUT:

- `n` - C long

OUTPUT:

- `gen` - PARI list of first n primes

EXAMPLES:

```
sage: pari.prime_list(0)
[]
sage: pari.prime_list(-1)
[]
sage: pari.prime_list(3)
[2, 3, 5]
sage: pari.prime_list(10)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
sage: pari.prime_list(20)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
```

```
sage: len(pari.prime_list(1000))
1000
```

primes_up_to_n(*n*)

Return the primes <= n as a pari list.

EXAMPLES:

```
sage: pari.primes_up_to_n(1)
[]
sage: pari.primes_up_to_n(20)
[2, 3, 5, 7, 11, 13, 17, 19]
```

read(*filename*)

Read a script from the named filename into the interpreter. The functions defined in the script are then available for use from Sage/PARI. The result of the last command in `filename` is returned.

EXAMPLES:

Create a gp file:

```
sage: import tempfile
sage: gpfile = tempfile.NamedTemporaryFile(mode="w")
sage: gpfile.file.write("mysquare(n) = {\n")
sage: gpfile.file.write("    n^2;\n")
sage: gpfile.file.write("}\n")
sage: gpfile.file.write("polcyclo(5)\n")
sage: gpfile.file.flush()
```

Read it in Sage, we get the result of the last line:

```
sage: pari.read(gpfile.name)
x^4 + x^3 + x^2 + x + 1
```

Call the function defined in the gp file:

```
sage: pari('mysquare(12)')
144
```

set_debug_level(*level*)

Set the debug PARI C library variable.

set_real_precision(*n*)

Sets the PARI default real precision in decimal digits.

This is used both for creation of new objects from strings and for printing. It is the number of digits *IN DECIMAL* in which real numbers are printed. It also determines the precision of objects created by parsing strings (e.g. pari('1.2')), which is *not* the normal way of creating new pari objects in Sage. It has *no* effect on the precision of computations within the pari library.

Returns the previous PARI real precision.

EXAMPLES:

```
sage: pari.set_real_precision(60)
15
sage: pari('1.2')
1.20000000000000000000000000000000000000000000000000000000000
sage: pari.set_real_precision(15)
60
```

set_series_precision(*n*)

setrand(*seed*)

Sets PARI's current random number seed.

INPUT:

- `seed` – either a strictly positive integer or a GEN of type `t_VECSMALL` as output by `getrand()`

This should not be called directly; instead, use Sage's global random number seed handling in `sage.misc.randstate` and call `current_randstate().set_seed_pari()`.

EXAMPLES:

```
sage: pari.setrand(50)
sage: a = pari.getrand(); a
Vecsmall([...])
sage: pari.setrand(a)
sage: a == pari.getrand()
True
```

TESTS:

Check that invalid inputs are handled properly (#11825):

```
sage: pari.setrand(0)
Traceback (most recent call last):
...
PariError: incorrect type in setrand
sage: pari.setrand("foobar")
Traceback (most recent call last):
...
PariError: incorrect type in setrand
```

stacksize()

Returns the current size of the PARI stack, which is 10^6 by default. However, the stack size is automatically doubled when needed. It can also be set directly using `pari.allocatemem()`.

EXAMPLES:

```
sage: pari.stacksize()
1000000
```

vector(*n*, *entries=None*)

vector(long n, entries=None): Create and return the length n PARI vector with given list of entries.

EXAMPLES:

```
sage: pari.vector(5, [1, 2, 5, 4, 3])
[1, 2, 5, 4, 3]
sage: pari.vector(2, [x, 1])
[x, 1]
sage: pari.vector(2, [x, 1, 5])
Traceback (most recent call last):
...
IndexError: length of entries (=3) must equal n (=2)
```

`sage.libs.pari.pari_instance.`**prec_bits_to_dec**(*prec_in_bits*)

Convert from precision expressed in bits to precision expressed in decimal.

EXAMPLES:

```
sage: from sage.libs.pari.pari_instance import prec_bits_to_dec
sage: prec_bits_to_dec(53)
15
```

```
sage: [(32*n, prec_bits_to_dec(32*n)) for n in range(1, 9)]
[(32, 9),
(64, 19),
(96, 28),
(128, 38),
(160, 48),
(192, 57),
(224, 67),
(256, 77)]
```

sage.libs.pari.pari_instance.**prec_bits_to_words**(*prec_in_bits*)

Convert from precision expressed in bits to pari real precision expressed in words. Note: this rounds up to the nearest word, adjusts for the two codewords of a pari real, and is architecture-dependent.

EXAMPLES:

```
sage: from sage.libs.pari.pari_instance import prec_bits_to_words
sage: prec_bits_to_words(70)
5   # 32-bit
4   # 64-bit

sage: [(32*n, prec_bits_to_words(32*n)) for n in range(1, 9)]
[(32, 3), (64, 4), (96, 5), (128, 6), (160, 7), (192, 8), (224, 9), (256, 10)] # 32-bit
[(32, 3), (64, 3), (96, 4), (128, 4), (160, 5), (192, 5), (224, 6), (256, 6)] # 64-bit
```

sage.libs.pari.pari_instance.**prec_dec_to_bits**(*prec_in_dec*)

Convert from precision expressed in decimal to precision expressed in bits.

EXAMPLES:

```
sage: from sage.libs.pari.pari_instance import prec_dec_to_bits
sage: prec_dec_to_bits(15)
49
sage: [(n, prec_dec_to_bits(n)) for n in range(10, 100, 10)]
[(10, 33),
(20, 66),
(30, 99),
(40, 132),
(50, 166),
(60, 199),
(70, 232),
(80, 265),
(90, 298)]
```

sage.libs.pari.pari_instance.**prec_dec_to_words**(*prec_in_dec*)

Convert from precision expressed in decimal to precision expressed in words. Note: this rounds up to the nearest word, adjusts for the two codewords of a pari real, and is architecture-dependent.

EXAMPLES:

```
sage: from sage.libs.pari.pari_instance import prec_dec_to_words
sage: prec_dec_to_words(38)
6   # 32-bit
4   # 64-bit
sage: [(n, prec_dec_to_words(n)) for n in range(10, 90, 10)]
[(10, 4), (20, 5), (30, 6), (40, 7), (50, 8), (60, 9), (70, 10), (80, 11)] # 32-bit
[(10, 3), (20, 4), (30, 4), (40, 5), (50, 5), (60, 6), (70, 6), (80, 7)] # 64-bit
```

sage.libs.pari.pari_instance.**prec_words_to_bits**(*prec_in_words*)

Convert from pari real precision expressed in words to precision expressed in bits. Note: this adjusts for the two codewords of a pari real, and is architecture-dependent.

EXAMPLES:

```
sage: from sage.libs.pari.pari_instance import prec_words_to_bits
sage: prec_words_to_bits(10)
256   # 32-bit
512   # 64-bit
sage: [(n, prec_words_to_bits(n)) for n in range(3, 10)]
[(3, 32), (4, 64), (5, 96), (6, 128), (7, 160), (8, 192), (9, 224)]  # 32-bit
[(3, 64), (4, 128), (5, 192), (6, 256), (7, 320), (8, 384), (9, 448)] # 64-bit
```

sage.libs.pari.pari_instance.**prec_words_to_dec**(*prec_in_words*)

Convert from precision expressed in words to precision expressed in decimal. Note: this adjusts for the two codewords of a pari real, and is architecture-dependent.

EXAMPLES:

```
sage: from sage.libs.pari.pari_instance import prec_words_to_dec
sage: prec_words_to_dec(5)
28   # 32-bit
57   # 64-bit
sage: [(n, prec_words_to_dec(n)) for n in range(3, 10)]
[(3, 9), (4, 19), (5, 28), (6, 38), (7, 48), (8, 57), (9, 67)] # 32-bit
[(3, 19), (4, 38), (5, 57), (6, 77), (7, 96), (8, 115), (9, 134)] # 64-bit
```

CHAPTER

THIRTEEN

SAGE CLASS FOR PARI'S GEN TYPE

Sage class for PARI's GEN type

See the `PariInstance` class for documentation and examples.

AUTHORS:

- William Stein (2006-03-01): updated to work with PARI 2.2.12-beta
- William Stein (2006-03-06): added newtonpoly
- Justin Walker: contributed some of the function definitions
- Gonzalo Tornaria: improvements to conversions; much better error handling.
- Robert Bradshaw, Jeroen Demeyer, William Stein (2010-08-15): Upgrade to PARI 2.4.3 (#9343)
- Jeroen Demeyer (2011-11-12): rewrite various conversion routines (#11611, #11854, #11952)
- Peter Bruin (2013-11-17): move PariInstance to a separate file (#15185)

exception `sage.libs.pari.gen.`**`PariError`**

Bases: `exceptions.RuntimeError`

Error raised by PARI

`errnum``()`

Return the PARI error number corresponding to this exception.

EXAMPLES:

```
sage: try:
....:     pari('1/0')
....: except PariError as err:
....:     print err.errnum()
27
```

`errtext``()`

Return the message output by PARI when this error occurred.

EXAMPLE:

```
sage: try:
....:     pari('pi()')
....: except PariError as e:
....:     print e.errtext()
....:
  ***   at top-level: pi()
  ***                 ^----
  ***   not a function in function call
```

class `sage.libs.pari.gen.`**`gen`**

Bases: `sage.structure.element.RingElement`

Python extension class that models the PARI GEN type.

`Col` (*x*, *n=0*)

Transform the object x into a column vector with minimal size $|n|$.

INPUT:

- `x` – gen
- `n` – Make the column vector of minimal length $|n|$. If $n > 0$, append zeros; if $n < 0$, prepend zeros.

OUTPUT:

A PARI column vector (type `t_COL`)

EXAMPLES:

```
sage: pari(1.5).Col()
[1.50000000000000]~
sage: pari([1,2,3,4]).Col()
[1, 2, 3, 4]~
sage: pari('[1,2; 3,4]').Col()
[[1, 2], [3, 4]]~
sage: pari('"Sage"').Col()
["S", "a", "g", "e"]~
sage: pari('x + 3*x^3').Col()
[3, 0, 1, 0]~
sage: pari('x + 3*x^3 + O(x^5)').Col()
[1, 0, 3, 0]~
```

We demonstate the n argument:

```
sage: pari([1,2,3,4]).Col(2)
[1, 2, 3, 4]~
sage: pari([1,2,3,4]).Col(-2)
[1, 2, 3, 4]~
sage: pari([1,2,3,4]).Col(6)
[1, 2, 3, 4, 0, 0]~
sage: pari([1,2,3,4]).Col(-6)
[0, 0, 1, 2, 3, 4]~
```

See also `Vec()` (create a row vector) for more examples and `Colrev()` (create a column in reversed order).

`Colrev` (*x*, *n=0*)

Transform the object x into a column vector with minimal size $|n|$. The order of the resulting vector is reversed compared to `Col()`.

INPUT:

- `x` – gen
- `n` – Make the vector of minimal length $|n|$. If $n > 0$, prepend zeros; if $n < 0$, append zeros.

OUTPUT:

A PARI column vector (type `t_COL`)

EXAMPLES:

```
sage: pari(1.5).Colrev()
[1.50000000000000]~
```

```
sage: pari([1,2,3,4]).Colrev()
[4, 3, 2, 1]~
sage: pari('[1,2; 3,4]').Colrev()
[[3, 4], [1, 2]]~
sage: pari('x + 3*x^3').Colrev()
[0, 1, 0, 3]~
```

We demonstate the n argument:

```
sage: pari([1,2,3,4]).Colrev(2)
[4, 3, 2, 1]~
sage: pari([1,2,3,4]).Colrev(-2)
[4, 3, 2, 1]~
sage: pari([1,2,3,4]).Colrev(6)
[0, 0, 4, 3, 2, 1]~
sage: pari([1,2,3,4]).Colrev(-6)
[4, 3, 2, 1, 0, 0]~
```

List (*x*)

List(x): transforms the PARI vector or list x into a list.

EXAMPLES:

```
sage: v = pari([1,2,3])
sage: v
[1, 2, 3]
sage: v.type()
't_VEC'
sage: w = v.List()
sage: w
List([1, 2, 3])
sage: w.type()
't_LIST'
```

Mat (*x*)

Mat(x): Returns the matrix defined by x.

- If x is already a matrix, a copy of x is created and returned.
- If x is not a vector or a matrix, this function returns a 1x1 matrix.
- If x is a row (resp. column) vector, this functions returns a 1-row (resp. 1-column) matrix, *unless* all elements are column (resp. row) vectors of the same length, in which case the vectors are concatenated sideways and the associated big matrix is returned.

INPUT:

- x - gen

OUTPUT:

- `gen` - a PARI matrix

EXAMPLES:

```
sage: x = pari(5)
sage: x.type()
't_INT'
sage: y = x.Mat()
sage: y
Mat(5)
sage: y.type()
```

```
't_MAT'
sage: x = pari('[1,2;3,4]')
sage: x.type()
't_MAT'
sage: x = pari('[1,2,3,4]')
sage: x.type()
't_VEC'
sage: y = x.Mat()
sage: y
Mat([1, 2, 3, 4])
sage: y.type()
't_MAT'

sage: v = pari('[1,2;3,4]').Vec(); v
[[1, 3]~, [2, 4]~]
sage: v.Mat()
[1, 2; 3, 4]
sage: v = pari('[1,2;3,4]').Col(); v
[[1, 2], [3, 4]]~
sage: v.Mat()
[1, 2; 3, 4]
```

Mod (*x*, *y*)

Mod(x, y): Returns the object x modulo y, denoted Mod(x, y).

The input y must be a an integer or a polynomial:

- If y is an INTEGER, x must also be an integer, a rational number, or a p-adic number compatible with the modulus y.
- If y is a POLYNOMIAL, x must be a scalar (which is not a polmod), a polynomial, a rational function, or a power series.

Warning: This function is not the same as `x % y` which is an integer or a polynomial.

INPUT:

- `x` - gen
- `y` - integer or polynomial

OUTPUT:

- `gen` - intmod or polmod

EXAMPLES:

```
sage: z = pari(3)
sage: x = z.Mod(pari(7))
sage: x
Mod(3, 7)
sage: x^2
Mod(2, 7)
sage: x^100
Mod(4, 7)
sage: x.type()
't_INTMOD'

sage: f = pari("x^2 + x + 1")
sage: g = pari("x")
```

```
sage: a = g.Mod(f)
sage: a
Mod(x, x^2 + x + 1)
sage: a*a
Mod(-x - 1, x^2 + x + 1)
sage: a.type()
't_POLMOD'
```

Pol(*v=-1*)

Pol(x, v): convert x into a polynomial with main variable v and return the result.

- If x is a scalar, returns a constant polynomial.
- If x is a power series, the effect is identical to `truncate`, i.e. it chops off the $O(X^k)$.
- If x is a vector, this function creates the polynomial whose coefficients are given in x, with x[0] being the leading coefficient (which can be zero).

Warning: This is *not* a substitution function. It will not transform an object containing variables of higher priority than v:

```
sage: pari('x+y').Pol('y')
Traceback (most recent call last):
...
PariError: variable must have higher priority in gtopoly
```

INPUT:

- x - gen
- v - (optional) which variable, defaults to 'x'

OUTPUT:

- `gen` - a polynomial

EXAMPLES:

```
sage: v = pari("[1,2,3,4]")
sage: f = v.Pol()
sage: f
x^3 + 2*x^2 + 3*x + 4
sage: f*f
x^6 + 4*x^5 + 10*x^4 + 20*x^3 + 25*x^2 + 24*x + 16

sage: v = pari("[1,2;3,4]")
sage: v.Pol()
[1, 3]~*x + [2, 4]~
```

Polrev(*v=-1*)

Polrev(x, v): Convert x into a polynomial with main variable v and return the result. This is the reverse of Pol if x is a vector, otherwise it is identical to Pol. By "reverse" we mean that the coefficients are reversed.

INPUT:

- x - gen

OUTPUT:

- `gen` - a polynomial

EXAMPLES:

```
sage: v = pari("[1,2,3,4]")
sage: f = v.Polrev()
sage: f
4*x^3 + 3*x^2 + 2*x + 1
sage: v.Pol()
x^3 + 2*x^2 + 3*x + 4
sage: v.Polrev('y')
4*y^3 + 3*y^2 + 2*y + 1
```

Note that Polrev does *not* reverse the coefficients of a polynomial!

```
sage: f
4*x^3 + 3*x^2 + 2*x + 1
sage: f.Polrev()
4*x^3 + 3*x^2 + 2*x + 1
sage: v = pari("[1,2;3,4]")
sage: v.Polrev()
[2, 4]~*x + [1, 3]~
```

Qfb (*a*, *b*, *c*, *D=0*, *precision=0*)

Qfb(a,b,c,D=0.): Returns the binary quadratic form

$$ax^2 + bxy + cy^2.$$

The optional D is 0 by default and initializes Shank's distance if $b^2 - 4ac > 0$. The discriminant of the quadratic form must not be a perfect square.

Note: Negative definite forms are not implemented, so use their positive definite counterparts instead. (I.e., if f is a negative definite quadratic form, then -f is positive definite.)

INPUT:

- `a` - gen
- `b` - gen
- `c` - gen
- `D` - gen (optional, defaults to 0)

OUTPUT:

- `gen` - binary quadratic form

EXAMPLES:

```
sage: pari(3).Qfb(7, 1)
Qfb(3, 7, 1, 0.E-19)
sage: pari(3).Qfb(7, 2)  # discriminant is 25
Traceback (most recent call last):
...
PariError: square discriminant in Qfb
```

Ser (*f*, *v=-1*, *precision=-1*)

Return a power series or Laurent series in the variable v constructed from the object f.

INPUT:

- `f` – PARI gen

- `v` – PARI variable (default: x)
- `precision` – the desired relative precision (default: the value returned by `pari.get_series_precision()`). This is the absolute precision minus the v-adic valuation.

OUTPUT:

- PARI object of type `t_SER`

The series is constructed from f in the following way:

- If f is a scalar, a constant power series is returned.
- If f is a polynomial, it is converted into a power series in the obvious way.
- If f is a rational function, it will be expanded in a Laurent series around $v = 0$.
- If f is a vector, its coefficients become the coefficients of the power series, starting from the constant term. This is the convention used by the function `Polrev()`, and the reverse of that used by `Pol()`.

Warning: This function will not transform objects containing variables of higher priority than v.

EXAMPLES:

```
sage: pari(2).Ser()
2 + O(x^16)
sage: pari(Mod(0, 7)).Ser()
O(x^16)

sage: x = pari([1, 2, 3, 4, 5])
sage: x.Ser()
1 + 2*x + 3*x^2 + 4*x^3 + 5*x^4 + O(x^16)
sage: f = x.Ser('v'); print f
1 + 2*v + 3*v^2 + 4*v^3 + 5*v^4 + O(v^16)
sage: pari(1)/f
1 - 2*v + v^2 + 6*v^5 - 17*v^6 + 16*v^7 - 5*v^8 + 36*v^10 - 132*v^11 + 181*v^12 - 110*v^13 +

sage: pari('x^5').Ser(precision=20)
x^5 + O(x^25)
sage: pari('1/x').Ser(precision=1)
x^-1 + O(x^0)
```

Set (*x*)

Set(x): convert x into a set, i.e. a row vector of strings in increasing lexicographic order.

INPUT:

- x - gen

OUTPUT:

- `gen` - a vector of strings in increasing lexicographic order.

EXAMPLES:

```
sage: pari([1,5,2]).Set()
["1", "2", "5"]
sage: pari([]).Set()       # the empty set
[]
sage: pari([1,1,-1,-1,3,3]).Set()
["-1", "1", "3"]
sage: pari(1).Set()
```

```
["1"]
sage: pari('1/(x*y)').Set()
["1/(y*x)"]
sage: pari('["bc","ab","bc"]').Set()
[""ab"", ""bc""]
```

Str()

Str(self): Return the print representation of self as a PARI object.

INPUT:

- `self` - gen

OUTPUT:

- `gen` - a PARI gen of type t_STR, i.e., a PARI string

EXAMPLES:

```
sage: pari([1,2,['abc',1]]).Str()
"[1, 2, [abc, 1]]"
sage: pari([1,1, 1.54]).Str()
"[1, 1, 1.54000000000000]"
sage: pari(1).Str()       # 1 is automatically converted to string rep
"1"
sage: x = pari('x')       # PARI variable "x"
sage: x.Str()             # is converted to string rep.
"x"
sage: x.Str().type()
't_STR'
```

Strchr(*x*)

Strchr(x): converts x to a string, translating each integer into a character (in ASCII).

Note: `Vecsmall()` is (essentially) the inverse to `Strchr()`.

INPUT:

- `x` - PARI vector of integers

OUTPUT:

- `gen` - a PARI string

EXAMPLES:

```
sage: pari([65,66,123]).Strchr()
"AB{"
sage: pari('"Sage"').Vecsmall()   # pari('"Sage"') --> PARI t_STR
Vecsmall([83, 97, 103, 101])
sage: _.Strchr()
"Sage"
sage: pari([83, 97, 103, 101]).Strchr()
"Sage"
```

Strexpand(*x*)

Strexpand(x): Concatenate the entries of the vector x into a single string, performing tilde expansion.

Note: I have no clue what the point of this function is. - William

Strtex(*x*)

Strtex(x): Translates the vector x of PARI gens to TeX format and returns the resulting concatenated strings as a PARI t_STR.

INPUT:

- `x` - gen

OUTPUT:

- `gen` - PARI t_STR (string)

EXAMPLES:

```
sage: v=pari('x^2')
sage: v.Strtex()
"x^2"
sage: v=pari(['1/x^2','x'])
sage: v.Strtex()
"\\frac{1}{x^2}x"
sage: v=pari(['1 + 1/x + 1/(y+1)','x-1'])
sage: v.Strtex()
"\\frac{ \\left(y\n + 2\\right)  x\n + \\left(y\n + 1\\right) }{ \\left(y\n + 1\\right)  x}x
```

Vec(*x*, *n=0*)

Transform the object x into a vector with minimal size $|n|$.

INPUT:

- `x` – gen
- `n` – Make the vector of minimal length $|n|$. If $n > 0$, append zeros; if $n < 0$, prepend zeros.

OUTPUT:

A PARI vector (type `t_VEC`)

EXAMPLES:

```
sage: pari(1).Vec()
[1]
sage: pari('x^3').Vec()
[1, 0, 0, 0]
sage: pari('x^3 + 3*x - 2').Vec()
[1, 0, 3, -2]
sage: pari([1,2,3]).Vec()
[1, 2, 3]
sage: pari('[1, 2; 3, 4]').Vec()
[[1, 3]~, [2, 4]~]
sage: pari('"Sage"').Vec()
["S", "a", "g", "e"]
sage: pari('2*x^2 + 3*x^3 + O(x^5)').Vec()
[2, 3, 0]
sage: pari('2*x^-2 + 3*x^3 + O(x^5)').Vec()
[2, 0, 0, 0, 0, 3, 0]
```

Note the different term ordering for polynomials and series:

```
sage: pari('1 + x + 3*x^3 + O(x^5)').Vec()
[1, 1, 0, 3, 0]
sage: pari('1 + x + 3*x^3').Vec()
[3, 0, 1, 1]
```

We demonstate the n argument:

```
sage: pari([1,2,3,4]).Vec(2)
[1, 2, 3, 4]
sage: pari([1,2,3,4]).Vec(-2)
[1, 2, 3, 4]
sage: pari([1,2,3,4]).Vec(6)
[1, 2, 3, 4, 0, 0]
sage: pari([1,2,3,4]).Vec(-6)
[0, 0, 1, 2, 3, 4]
```

See also `Col()` (create a column vector) and `Vecrev()` (create a vector in reversed order).

Vecrev (*x*, *n=0*)

Transform the object x into a vector with minimal size $|n|$. The order of the resulting vector is reversed compared to `Vec()`.

INPUT:

- `x` – gen
- `n` – Make the vector of minimal length $|n|$. If $n > 0$, prepend zeros; if $n < 0$, append zeros.

OUTPUT:

A PARI vector (type `t_VEC`)

EXAMPLES:

```
sage: pari(1).Vecrev()
[1]
sage: pari('x^3').Vecrev()
[0, 0, 0, 1]
sage: pari('x^3 + 3*x - 2').Vecrev()
[-2, 3, 0, 1]
sage: pari([1, 2, 3]).Vecrev()
[3, 2, 1]
sage: pari('Col([1, 2, 3])').Vecrev()
[3, 2, 1]
sage: pari('[1, 2; 3, 4]').Vecrev()
[[2, 4]~, [1, 3]~]
sage: pari('"Sage"').Vecrev()
["e", "g", "a", "S"]
```

We demonstate the n argument:

```
sage: pari([1,2,3,4]).Vecrev(2)
[4, 3, 2, 1]
sage: pari([1,2,3,4]).Vecrev(-2)
[4, 3, 2, 1]
sage: pari([1,2,3,4]).Vecrev(6)
[0, 0, 4, 3, 2, 1]
sage: pari([1,2,3,4]).Vecrev(-6)
[4, 3, 2, 1, 0, 0]
```

Vecsmall (*x*, *n=0*)

Transform the object x into a `t_VECSMALL` with minimal size $|n|$.

INPUT:

- `x` – gen
- `n` – Make the vector of minimal length $|n|$. If $n > 0$, append zeros; if $n < 0$, prepend zeros.

OUTPUT:

A PARI vector of small integers (type `t_VECSMALL`)

EXAMPLES:

```
sage: pari([1,2,3]).Vecsmall()
Vecsmall([1, 2, 3])
sage: pari('"Sage"').Vecsmall()
Vecsmall([83, 97, 103, 101])
sage: pari(1234).Vecsmall()
Vecsmall([1234])
sage: pari('x^2 + 2*x + 3').Vecsmall()
Traceback (most recent call last):
...
PariError: incorrect type in vectosmall
```

We demonstate the n argument:

```
sage: pari([1,2,3]).Vecsmall(2)
Vecsmall([1, 2, 3])
sage: pari([1,2,3]).Vecsmall(-2)
Vecsmall([1, 2, 3])
sage: pari([1,2,3]).Vecsmall(6)
Vecsmall([1, 2, 3, 0, 0, 0])
sage: pari([1,2,3]).Vecsmall(-6)
Vecsmall([0, 0, 0, 1, 2, 3])
```

Zn_issquare (*n*)

Return `True` if `self` is a square modulo n, `False` if not.

INPUT:

- `self` – integer
- `n` – integer or factorisation matrix

EXAMPLES:

```
sage: pari(3).Zn_issquare(4)
False
sage: pari(4).Zn_issquare(30.factor())
True
```

Zn_sqrt (*n*)

Return a square root of `self` modulo n, if such a square root exists; otherwise, raise a `ValueError`.

INPUT:

- `self` – integer
- `n` – integer or factorisation matrix

EXAMPLES:

```
sage: pari(3).Zn_sqrt(4)
Traceback (most recent call last):
...
ValueError: 3 is not a square modulo 4
sage: pari(4).Zn_sqrt(30.factor())
22
```

abs (*x*, *precision=0*)

Returns the absolute value of x (its modulus, if x is complex). Rational functions are not allowed. Contrary to most transcendental functions, an exact argument is not converted to a real number before applying abs and an exact result is returned if possible.

EXAMPLES:

```
sage: x = pari("-27.1")
sage: x.abs()
27.1000000000000
sage: pari('1 + I').abs(precision=128).sage()
1.4142135623730950488016887242096980786
```

If x is a polynomial, returns -x if the leading coefficient is real and negative else returns x. For a power series, the constant coefficient is considered instead.

EXAMPLES:

```
sage: pari('x-1.2*x^2').abs()
1.20000000000000*x^2 - x
sage: pari('-2 + t + O(t^2)').abs()
2 - t + O(t^2)
```

acos (*x*, *precision=0*)

The principal branch of $\cos^{-1}(x)$, so that $\mathbf{R}e(\mathrm{acos}(x))$ belongs to $[0, Pi]$. If x is real and $|x| > 1$, then $\mathrm{acos}(x)$ is complex.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(0.5).acos()
1.04719755119660
sage: pari(1/2).acos()
1.04719755119660
sage: pari(1.1).acos()
0.443568254385115*I
sage: C.<i> = ComplexField()
sage: pari(1.1+i).acos()
0.849343054245252 - 1.09770986682533*I
```

acosh (*x*, *precision=0*)

The principal branch of $\cosh^{-1}(x)$, so that $\Im(\mathrm{acosh}(x))$ belongs to $[0, Pi]$. If x is real and $x < 1$, then $\mathrm{acosh}(x)$ is complex.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).acosh()
1.31695789692482
sage: pari(0).acosh()
1.57079632679490*I
sage: C.<i> = ComplexField()
sage: pari(i).acosh()
0.881373587019543 + 1.57079632679490*I
```

agm (*x*, *y*, *precision=0*)

The arithmetic-geometric mean of x and y. In the case of complex or negative numbers, the principal square root is always chosen. p-adic or power series arguments are also allowed. Note that a p-adic AGM exists only if x/y is congruent to 1 modulo p (modulo 16 for p=2). x and y cannot both be vectors or matrices.

If any of x or y is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their two precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).agm(2)
2.00000000000000
sage: pari(0).agm(1)
0
sage: pari(1).agm(2)
1.45679103104691
sage: C.<i> = ComplexField()
sage: pari(1+i).agm(-3)
-0.964731722290876 + 1.15700282952632*I
```

algdep(*n*)

EXAMPLES:

```
sage: n = pari.set_real_precision(210)
sage: w1 = pari('z1=2-sqrt(26); (z1+I)/(z1-I)')
sage: f = w1.algdep(12); f
545*x^11 - 297*x^10 - 281*x^9 + 48*x^8 - 168*x^7 + 690*x^6 - 168*x^5 + 48*x^4 - 281*x^3 - 29
sage: f(w1).abs() < 1.0e-200
True
sage: f.factor()
[x, 1; x + 1, 2; x^2 + 1, 1; x^2 + x + 1, 1; 545*x^4 - 1932*x^3 + 2790*x^2 - 1932*x + 545, 1
sage: pari.set_real_precision(n)
210
```

arg(*x*, *precision=0*)

arg(x): argument of x,such that $-\pi < \arg(x) \leq \pi$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: C.<i> = ComplexField()
sage: pari(2+i).arg()
0.463647609000806
```

asin(*x*, *precision=0*)

The principal branch of $\sin^{-1}(x)$, so that $\mathbf{R}e(\operatorname{asin}(x))$ belongs to $[-\pi/2, \pi/2]$. If x is real and $|x| > 1$ then $\operatorname{asin}(x)$ is complex.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(pari(0.5).sin()).asin()
0.500000000000000
sage: pari(2).asin()
1.57079632679490 - 1.31695789692482*I
```

asinh (*x*, *precision=0*)

The principal branch of $\sinh^{-1}(x)$, so that $\Im(\text{asinh}(x))$ belongs to $[-\pi/2, \pi/2]$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).asinh()
1.44363547517881
sage: C.<i> = ComplexField()
sage: pari(2+i).asinh()
1.52857091948100 + 0.427078586392476*I
```

atan (*x*, *precision=0*)

The principal branch of $\tan^{-1}(x)$, so that $\mathbf{R}e(\text{atan}(x))$ belongs to $]-\pi/2, \pi/2[$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).atan()
0.785398163397448
sage: C.<i> = ComplexField()
sage: pari(1.5+i).atan()
1.10714871779409 + 0.255412811882995*I
```

atanh (*x*, *precision=0*)

The principal branch of $\tanh^{-1}(x)$, so that $\Im(\text{atanh}(x))$ belongs to $]-\pi/2, \pi/2]$. If x is real and $|x| > 1$ then $\text{atanh}(x)$ is complex.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(0).atanh()
0.E-19
sage: pari(2).atanh()
0.549306144334055 - 1.57079632679490*I
```

bernfrac (*x*)

The Bernoulli number B_x, where $B_0 = 1$, $B_1 = -1/2$, $B_2 = 1/6, \ldots,$ expressed as a rational number. The argument x should be of type integer.

EXAMPLES:

```
sage: pari(18).bernfrac()
43867/798
sage: [pari(n).bernfrac() for n in range(10)]
[1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, -1/30, 0]
```

bernreal (*x*, *precision=0*)

The Bernoulli number B_x, as for the function bernfrac, but B_x is returned as a real number (with the current precision).

EXAMPLES:

```
sage: pari(18).bernreal()
54.9711779448622
sage: pari(18).bernreal(precision=192).sage()
54.9711779448621553884711779448621553884711779448621553885
```

bernvec(*x*)

Creates a vector containing, as rational numbers, the Bernoulli numbers $B_0, B_2, \ldots, B_{2x}$. This routine is obsolete. Use bernfrac instead each time you need a Bernoulli number in exact form.

Note: this routine is implemented using repeated independent calls to bernfrac, which is faster than the standard recursion in exact arithmetic.

EXAMPLES:

```
sage: pari(8).bernvec()
[1, 1/6, -1/30, 1/42, -1/30, 5/66, -691/2730, 7/6, -3617/510]
sage: [pari(2*n).bernfrac() for n in range(9)]
[1, 1/6, -1/30, 1/42, -1/30, 5/66, -691/2730, 7/6, -3617/510]
```

besselh1(*nu*, *x*, *precision=0*)

The H^1-Bessel function of index ν and argument x.

If nu or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).besselh1(3)
0.486091260585891 - 0.160400393484924*I
```

besselh2(*nu*, *x*, *precision=0*)

The H^2-Bessel function of index ν and argument x.

If nu or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).besselh2(3)
0.486091260585891 + 0.160400393484924*I
```

besseli(*nu*, *x*, *precision=0*)

Bessel I function (Bessel function of the second kind), with index ν and argument x. If x converts to a power series, the initial factor $(x/2)^\nu/\Gamma(\nu+1)$ is omitted (since it cannot be represented in PARI when ν is not integral).

If nu or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).besseli(3)
2.24521244092995
sage: C.<i> = ComplexField()
sage: pari(2).besseli(3+i)
1.12539407613913 + 2.08313822670661*I
```

besselj (*nu*, *x*, *precision=0*)

Bessel J function (Bessel function of the first kind), with index ν and argument x. If x converts to a power series, the initial factor $(x/2)^\nu/\Gamma(\nu+1)$ is omitted (since it cannot be represented in PARI when ν is not integral).

If nu or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).besselj(3)
0.486091260585891
```

besseljh (*nu*, *x*, *precision=0*)

J-Bessel function of half integral index (Spherical Bessel function of the first kind). More precisely, besseljh(n,x) computes $J_{n+1/2}(x)$ where n must an integer, and x is any complex value. In the current implementation (PARI, version 2.2.11), this function is not very accurate when x is small.

If nu or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).besseljh(3)
0.4127100324          # 32-bit
0.412710032209716     # 64-bit
```

besselk (*nu*, *x*, *flag=0*, *precision=0*)

nu.besselk(x, flag=0): K-Bessel function (modified Bessel function of the second kind) of index nu, which can be complex, and argument x.

If nu or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

INPUT:

- `nu` - a complex number
- `x` - real number (positive or negative)
- `flag` - default: 0 or 1: use hyperu (hyperu is much slower for small x, and doesn't work for negative x).

EXAMPLES:

```
sage: C.<i> = ComplexField()
sage: pari(2+i).besselk(3)
0.0455907718407551 + 0.0289192946582081*I

sage: pari(2+i).besselk(-3)
-4.34870874986752 - 5.38744882697109*I

sage: pari(2+i).besselk(300, flag=1)
3.74224603319728 E-132 + 2.49071062641525 E-134*I
```

besseln (*nu*, *x*, *precision=0*)

nu.besseln(x): Bessel N function (Spherical Bessel function of the second kind) of index nu and argument x.

If nu or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: C.<i> = ComplexField()
sage: pari(2+i).besseln(3)
-0.280775566958244 - 0.486708533223726*I
```

bezout (*x*, *y*)

bid_get_cyc ()

Returns the structure of the group $(O_K/I)^*$, where I is the ideal represented by `self`.

NOTE: `self` must be a "big ideal" (`bid`) as returned by `idealstar` for example.

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: J = pari(K).idealstar(K.ideal(4*i + 2))
sage: J.bid_get_cyc()
[4, 2]
```

bid_get_gen ()

Returns a vector of generators of the group $(O_K/I)^*$, where I is the ideal represented by `self`.

NOTE: `self` must be a "big ideal" (`bid`) with generators, as returned by `idealstar` with `flag` = 2.

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: J = pari(K).idealstar(K.ideal(4*i + 2), 2)
sage: J.bid_get_gen()
[7, [-2, -1]~]
```

We get an exception if we do not supply `flag = 2` to `idealstar`:

```
sage: J = pari(K).idealstar(K.ideal(3))
sage: J.bid_get_gen()
Traceback (most recent call last):
...
PariError: missing bid generators. Use idealstar(,,2)
```

binary (*x*)

binary(x): gives the vector formed by the binary digits of abs(x), where x is of type t_INT.

INPUT:

- x - gen of type t_INT

OUTPUT:

- `gen` - of type t_VEC

EXAMPLES:

```
sage: pari(0).binary()
[0]
sage: pari(-5).binary()
[1, 0, 1]
sage: pari(5).binary()
[1, 0, 1]
sage: pari(2005).binary()
[1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1]
```

```
sage: pari('"2"').binary()
Traceback (most recent call last):
...
TypeError: x (="2") must be of type t_INT, but is of type t_STR.
```

binomial (*x*, *k*)

binomial(x, k): return the binomial coefficient "x choose k".

INPUT:

•x - any PARI object (gen)

•k - integer

EXAMPLES:

```
sage: pari(6).binomial(2)
15
sage: pari('x+1').binomial(3)
1/6*x^3 - 1/6*x
sage: pari('2+x+O(x^2)').binomial(3)
1/3*x + O(x^2)
```

bitand (*x*, *y*)

bitand(x,y): Bitwise and of two integers x and y. Negative numbers behave as if modulo some large power of 2.

INPUT:

•x - gen (of type t_INT)

•y - coercible to gen (of type t_INT)

OUTPUT:

•gen - of type type t_INT

EXAMPLES:

```
sage: pari(8).bitand(4)
0
sage: pari(8).bitand(8)
8
sage: pari(6).binary()
[1, 1, 0]
sage: pari(7).binary()
[1, 1, 1]
sage: pari(6).bitand(7)
6
sage: pari(19).bitand(-1)
19
sage: pari(-1).bitand(-1)
-1
```

bitneg (*x*, *n=-1*)

bitneg(x,n=-1): Bitwise negation of the integer x truncated to n bits. n=-1 (the default) represents an infinite sequence of the bit 1. Negative numbers behave as if modulo some large power of 2.

With n=-1, this function returns -n-1. With n = 0, it returns a number a such that $a \cong -n - 1 \pmod{2^n}$.

INPUT:

•x - gen (t_INT)

- n - long, default = -1

OUTPUT:

- gen - t_INT

EXAMPLES:

```
sage: pari(10).bitneg()
-11
sage: pari(1).bitneg()
-2
sage: pari(-2).bitneg()
1
sage: pari(-1).bitneg()
0
sage: pari(569).bitneg()
-570
sage: pari(569).bitneg(10)
454
sage: 454 % 2^10
454
sage: -570 % 2^10
454
```

bitnegimply (*x*, *y*)

bitnegimply(x,y): Bitwise negated imply of two integers x and y, in other words, x BITAND BITNEG(y). Negative numbers behave as if modulo big power of 2.

INPUT:

- x - gen (of type t_INT)
- y - coercible to gen (of type t_INT)

OUTPUT:

- gen - of type type t_INT

EXAMPLES:

```
sage: pari(14).bitnegimply(0)
14
sage: pari(8).bitnegimply(8)
0
sage: pari(8+4).bitnegimply(8)
4
```

bitor (*x*, *y*)

bitor(x,y): Bitwise or of two integers x and y. Negative numbers behave as if modulo big power of 2.

INPUT:

- x - gen (of type t_INT)
- y - coercible to gen (of type t_INT)

OUTPUT:

- gen - of type type t_INT

EXAMPLES:

```
sage: pari(14).bitor(0)
14
```

```
sage: pari(8).bitor(4)
12
sage: pari(12).bitor(1)
13
sage: pari(13).bitor(1)
13
```

bittest (*x*, *n*)

bittest(x, long n): Returns bit number n (coefficient of 2^n in binary) of the integer x. Negative numbers behave as if modulo a big power of 2.

INPUT:

- x - gen (pari integer)

OUTPUT:

- bool - a Python bool

EXAMPLES:

```
sage: x = pari(6)
sage: x.bittest(0)
False
sage: x.bittest(1)
True
sage: x.bittest(2)
True
sage: x.bittest(3)
False
sage: pari(-3).bittest(0)
True
sage: pari(-3).bittest(1)
False
sage: [pari(-3).bittest(n) for n in range(10)]
[True, False, True, True, True, True, True, True, True, True]
```

bitxor (*x*, *y*)

bitxor(x,y): Bitwise exclusive or of two integers x and y. Negative numbers behave as if modulo big power of 2.

INPUT:

- x - gen (of type t_INT)
- y - coercible to gen (of type t_INT)

OUTPUT:

- gen - of type type t_INT

EXAMPLES:

```
sage: pari(6).bitxor(4)
2
sage: pari(0).bitxor(4)
4
sage: pari(6).bitxor(0)
6
```

bnf_get_cyc ()

Returns the structure of the class group of this number field as a vector of SNF invariants.

NOTE: `self` must be a “big number field” (`bnf`).

EXAMPLES:

```
sage: K.<a> = QuadraticField(-65)
sage: K.pari_bnf().bnf_get_cyc()
[4, 2]
```

bnf_get_gen()

Returns a vector of generators of the class group of this number field.

NOTE: `self` must be a “big number field” (`bnf`).

EXAMPLES:

```
sage: K.<a> = QuadraticField(-65)
sage: G = K.pari_bnf().bnf_get_gen(); G
[[3, 2; 0, 1], [2, 1; 0, 1]]
sage: map(lambda J: K.ideal(J), G)
[Fractional ideal (3, a + 2), Fractional ideal (2, a + 1)]
```

bnf_get_no()

Returns the class number of `self`, a “big number field” (`bnf`).

EXAMPLES:

```
sage: K.<a> = QuadraticField(-65)
sage: K.pari_bnf().bnf_get_no()
8
```

bnf_get_reg()

Returns the regulator of this number field.

NOTE: `self` must be a “big number field” (`bnf`).

EXAMPLES:

```
sage: K.<a> = NumberField(x^4 - 4*x^2 + 1)
sage: K.pari_bnf().bnf_get_reg()
2.66089858019037...
```

bnfcertify()

`bnf` being as output by `bnfinit`, checks whether the result is correct, i.e. whether the calculation of the contents of `self` are correct without assuming the Generalized Riemann Hypothesis. If it is correct, the answer is 1. If not, the program may output some error message or loop indefinitely.

For more information about PARI and the Generalized Riemann Hypothesis, see [PariUsers], page 120.

REFERENCES:

bnfinit(*flag=0*, *tech=None*, *precision=0*)

bnfisintnorm(*x*)

bnfisnorm(*x*, *flag=0*)

bnfisprincipal(*x*, *flag=1*)

bnfissunit(*sunit_data*, *x*)

bnfisunit(*x*)

bnfnarrow()

bnfsunit(*S*, *precision=0*)

bnfunit()

bnrclassno(*I*)

Return the order of the ray class group of self modulo I.

INPUT:

- self: a pari "BNF" object representing a number field
- I: a pari "BID" object representing an ideal of self

OUTPUT: integer

TESTS:

```
sage: K.<z> = QuadraticField(-23)
sage: p = K.primes_above(3)[0]
sage: K.pari_bnf().bnrclassno(p._pari_bid_())
3
```

ceil(*x*)

For real x: return the smallest integer = x. For rational functions: the quotient of numerator by denominator. For lists: apply componentwise.

INPUT:

- x - gen

OUTPUT:

- gen - depends on type of x

EXAMPLES:

```
sage: pari(1.4).ceil()
2
sage: pari(-1.4).ceil()
-1
sage: pari(3/4).ceil()
1
sage: pari(x).ceil()
x
sage: pari((x^2+x+1)/x).ceil()
x + 1
```

This may be unexpected: but it is correct, treating the argument as a rational function in RR(x).

```
sage: pari(x^2+5*x+2.5).ceil()
x^2 + 5*x + 2.50000000000000
```

centerlift(*x*, *v=-1*)

centerlift(x,v): Centered lift of x. This function returns exactly the same thing as lift, except if x is an integer mod.

INPUT:

- x - gen
- v - var (default: x)

OUTPUT: gen

EXAMPLES:

```
sage: x = pari(-2).Mod(5)
sage: x.centerlift()
-2
sage: x.lift()
3
sage: f = pari('x-1').Mod('x^2 + 1')
sage: f.centerlift()
x - 1
sage: f.lift()
x - 1
sage: f = pari('x-y').Mod('x^2+1')
sage: f
Mod(x - y, x^2 + 1)
sage: f.centerlift('x')
x - y
sage: f.centerlift('y')
Mod(x - y, x^2 + 1)
```

change_variable_name (*var*)

In `self`, which must be a `t_POL` or `t_SER`, set the variable to `var`. If the variable of `self` is already `var`, then return `self`.

Warning: You should be careful with variable priorities when applying this on a polynomial or series of which the coefficients have polynomial components. To be safe, only use this function on polynomials with integer or rational coefficients. For a safer alternative, use `subst()`.

EXAMPLES:

```
sage: f = pari('x^3 + 17*x + 3')
sage: f.change_variable_name("y")
y^3 + 17*y + 3
sage: f = pari('1 + 2*y + O(y^10)')
sage: f.change_variable_name("q")
1 + 2*q + O(q^10)
sage: f.change_variable_name("y") is f
True
```

In PARI, `I` refers to the square root of -1, so it cannot be used as variable name. Note the difference with `subst()`:

```
sage: f = pari('x^2 + 1')
sage: f.change_variable_name("I")
Traceback (most recent call last):
...
PariError: I already exists with incompatible valence
sage: f.subst("x", "I")
0
```

charpoly (*var=-1*, *flag=0*)

charpoly(A,v=x,flag=0): det(v*Id-A) = characteristic polynomial of A using the comatrix. flag is optional and may be set to 1 (use Lagrange interpolation) or 2 (use Hessenberg form), 0 being the default.

chinese (*y*)

component (*x*, *n*)

component(x, long n): Return n'th component of the internal representation of x. This function is 1-based instead of 0-based.

Note: For vectors or matrices, it is simpler to use x[n-1]. For list objects such as is output by nfinit, it is easier to use member functions.

INPUT:

- x - gen
- n - C long (coercible to)

OUTPUT: gen

EXAMPLES:

```
sage: pari([0,1,2,3,4]).component(1)
0
sage: pari([0,1,2,3,4]).component(2)
1
sage: pari([0,1,2,3,4]).component(4)
3
sage: pari('x^3 + 2').component(1)
2
sage: pari('x^3 + 2').component(2)
0
sage: pari('x^3 + 2').component(4)
1

sage: pari('x').component(0)
Traceback (most recent call last):
...
PariError: nonexistent component
```

concat (*y*)

conj (*x*)

conj(x): Return the algebraic conjugate of x.

INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari('x+1').conj()
x + 1
sage: pari('x+I').conj()
x - I
sage: pari('1/(2*x+3*I)').conj()
1/(2*x - 3*I)
sage: pari([1,2,'2-I','Mod(x,x^2+1)', 'Mod(x,x^2-2)']).conj()
[1, 2, 2 + I, Mod(-x, x^2 + 1), Mod(-x, x^2 - 2)]
sage: pari('Mod(x,x^2-2)').conj()
Mod(-x, x^2 - 2)
sage: pari('Mod(x,x^3-3)').conj()
Traceback (most recent call last):
...
PariError: incorrect type in gconj
```

conjvec (*x*, *precision=0*)

conjvec(x): Returns the vector of all conjugates of the algebraic number x. An algebraic number is a polynomial over Q modulo an irreducible polynomial.

INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari('Mod(1+x,x^2-2)').conjvec()
[-0.414213562373095, 2.41421356237310]~
sage: pari('Mod(x,x^3-3)').conjvec()
[1.44224957030741, -0.721124785153704 + 1.24902476648341*I, -0.721124785153704 - 1.24902476
sage: pari('Mod(1+x,x^2-2)').conjvec(precision=192)[0].sage()
-0.414213562373095048801688724209698078569671875376948073177
```

content()
Greatest common divisor of all the components of `self`.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: pari(2*x^2 + 2).content()
2
sage: pari("4*x^3 - 2*x/3 + 2/5").content()
2/15
```

contfrac(*x*, *b=0*, *lmax=0*)
contfrac(x,b,lmax): continued fraction expansion of x (x rational, real or rational function). b and lmax are both optional, where b is the vector of numerators of the continued fraction, and lmax is a bound for the number of terms in the continued fraction expansion.

contfracpnqn(*x*, *b=0*, *lmax=0*)
contfracpnqn(x): [p_n,p_n-1; q_n,q_n-1] corresponding to the continued fraction x.

cos(*x*, *precision=0*)
The cosine function.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1.5).cos()
0.0707372016677029
sage: C.<i> = ComplexField()
sage: pari(1+i).cos()
0.833730025131149 - 0.988897705762865*I
sage: pari('x+O(x^8)').cos()
1 - 1/2*x^2 + 1/24*x^4 - 1/720*x^6 + 1/40320*x^8 + O(x^9)
```

cosh(*x*, *precision=0*)
The hyperbolic cosine function.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1.5).cosh()
2.35240961524325
sage: C.<i> = ComplexField()
```

```
sage: pari(1+i).cosh()
0.833730025131149 + 0.988897705762865*I
sage: pari('x+O(x^8)').cosh()
1 + 1/2*x^2 + 1/24*x^4 + 1/720*x^6 + O(x^8)
```

cotan(*x*, *precision=0*)

The cotangent of x.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(5).cotan()
-0.295812915532746
```

Computing the cotangent of π doesn't raise an error, but instead just returns a very large (positive or negative) number.

```
sage: x = RR(pi)
sage: pari(x).cotan()         # random
-8.17674825 E15
```

debug(*depth=-1*)

Show the internal structure of self (like the `\x` command in gp).

EXAMPLE:

```
sage: pari('[1/2, 1.0*I]').debug()  # random addresses
[&=0000000004c5f010] VEC(lg=3):2200000000000003 0000000004c5eff8 0000000004c5efb0
  1st component = [&=0000000004c5eff8] FRAC(lg=3):0800000000000003 0000000004c5efe0 00000000
    num = [&=0000000004c5efe0] INT(lg=3):0200000000000003 (+,lgefint=3):4000000000000003 000
    den = [&=0000000004c5efc8] INT(lg=3):0200000000000003 (+,lgefint=3):4000000000000003 000
  2nd component = [&=0000000004c5efb0] COMPLEX(lg=3):0c00000000000003 00007fae8a2eb840 00000
    real = gen_0
    imag = [&=0000000004c5ef90] REAL(lg=4):0400000000000004 (+,expo=0):6000000000000000 8000
```

denominator(*x*)

denominator(x): Return the denominator of x. When x is a vector, this is the least common multiple of the denominators of the components of x.

what about poly? INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari('5/9').denominator()
9
sage: pari('(x+1)/(x-2)').denominator()
x - 2
sage: pari('2/3 + 5/8*x + 7/3*x^2 + 1/5*y').denominator()
1
sage: pari('2/3*x').denominator()
1
sage: pari('[2/3, 5/8, 7/3, 1/5]').denominator()
120
```

deriv(*v=-1*)

dilog(*x*, *precision=0*)

The principal branch of the dilogarithm of x, i.e. the analytic continuation of the power series $\log_2(x) = \sum_{n>=1} x^n/n^2$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).dilog()
1.64493406684823
sage: C.<i> = ComplexField()
sage: pari(1+i).dilog()
0.616850275068085 + 1.46036211675312*I
```

dirzetak(*n*)

disc()

e.disc(): return the discriminant of the elliptic curve e.

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.disc()
-161051
sage: _.factor()
[-1, 1; 11, 5]
```

divrem(*x*, *y*, *var=-1*)

divrem(x, y, v): Euclidean division of x by y giving as a 2-dimensional column vector the quotient and the remainder, with respect to v (to main variable if v is omitted).

eint1(*x*, *n=0*, *precision=0*)

x.eint1(n): exponential integral E1(x):

$$\int_x^\infty \frac{e^{-t}}{t} dt$$

If n is present, output the vector [eint1(x), eint1(2*x), ..., eint1(n*x)]. This is faster than repeatedly calling eint1(i*x).

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

REFERENCE:

- See page 262, Prop 5.6.12, of Cohen's book "A Course in Computational Algebraic Number Theory".

EXAMPLES:

elementval(*x*, *p*)

elladd(*z0*, *z1*)

e.elladd(z0, z1): return the sum of the points z0 and z1 on this elliptic curve.

INPUT:

- `e` - elliptic curve E
- `z0` - point on E
- `z1` - point on E

OUTPUT: point on E

EXAMPLES: First we create an elliptic curve:

```
sage: e = pari([0, 1, 1, -2, 0]).ellinit()
sage: str(e)[:65]   # first part of output
'[0, 1, 1, -2, 0, 4, -4, 1, -3, 112, -856, 389, 1404928/389, [0.90'
```

Next we add two points on the elliptic curve. Notice that the Python lists are automatically converted to PARI objects so you don't have to do that explicitly in your code.

```
sage: e.elladd([1,0], [-1,1])
[-3/4, -15/8]
```

ellak(*n*)

e.ellak(n): Returns the coefficient a_n of the L-function of the elliptic curve e, i.e. the n-th Fourier coefficient of the weight 2 newform associated to e (according to Shimura-Taniyama).

> The curve e *must* be a medium or long vector of the type given by ellinit. For this function to work for every n and not just those prime to the conductor, e must be a minimal Weierstrass equation. If this is not the case, use the function ellminimalmodel first before using ellak (or you will get INCORRECT RESULTS!)

INPUT:

- `e` - a PARI elliptic curve.
- `n` - integer.

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.ellak(6)
2
sage: e.ellak(2005)
2
sage: e.ellak(-1)
0
sage: e.ellak(0)
0
```

ellan(*n*, *python_ints=False*)

Return the first n Fourier coefficients of the modular form attached to this elliptic curve. See ellak for more details.

INPUT:

- `n` - a long integer
- `python_ints` - bool (default is False); if True, return a list of Python ints instead of a PARI gen wrapper.

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.ellan(3)
[1, -2, -1]
sage: e.ellan(20)
[1, -2, -1, 2, 1, 2, -2, 0, -2, -2, 1, -2, 4, 4, -1, -4, -2, 4, 0, 2]
sage: e.ellan(-1)
[]
sage: v = e.ellan(10, python_ints=True); v
[1, -2, -1, 2, 1, 2, -2, 0, -2, -2]
```

```
sage: type(v)
<type 'list'>
sage: type(v[0])
<type 'int'>
```

ellanalyticrank(*precision=0*)

Returns a 2-component vector with the order of vanishing at $s = 1$ of the L-function of the elliptic curve and the value of the first non-zero derivative.

EXAMPLE:

```
sage: E = EllipticCurve('389a1')
sage: pari(E).ellanalyticrank()
[2, 1.51863300057685]
```

ellap(*p*)

e.ellap(p): Returns the prime-indexed coefficient a_p of the L-function of the elliptic curve e, i.e. the p-th Fourier coefficient of the newform attached to e.

The computation uses the Shanks–Mestre method, or the SEA algorithm.

Warning: For this function to work for every n and not just those prime to the conductor, e must be a minimal Weierstrass equation. If this is not the case, use the function ellminimalmodel first before using ellap (or you will get INCORRECT RESULTS!)

INPUT:

- `e` - a PARI elliptic curve.
- `p` - prime integer

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.ellap(2)
-2
sage: e.ellap(2003)
4
sage: e.ellak(-1)
0
```

ellaplist(*n*, *python_ints=False*)

e.ellaplist(n): Returns a PARI list of all the prime-indexed coefficients a_p (up to n) of the L-function of the elliptic curve e, i.e. the Fourier coefficients of the newform attached to e.

INPUT:

- `self` – an elliptic curve
- `n` – a long integer
- `python_ints` – bool (default is False); if True, return a list of Python ints instead of a PARI gen wrapper.

Warning: The curve e must be a medium or long vector of the type given by ellinit. For this function to work for every n and not just those prime to the conductor, e must be a minimal Weierstrass equation. If this is not the case, use the function ellminimalmodel first before using ellaplist (or you will get INCORRECT RESULTS!)

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: v = e.ellaplist(10); v
[-2, -1, 1, -2]
sage: type(v)
<type 'sage.libs.pari.gen.gen'>
sage: v.type()
't_VEC'
sage: e.ellan(10)
[1, -2, -1, 2, 1, 2, -2, 0, -2, -2]
sage: v = e.ellaplist(10, python_ints=True); v
[-2, -1, 1, -2]
sage: type(v)
<type 'list'>
sage: type(v[0])
<type 'int'>
```

TESTS:

```
sage: v = e.ellaplist(1)
sage: print v, type(v)
[] <type 'sage.libs.pari.gen.gen'>
sage: v = e.ellaplist(1, python_ints=True)
sage: print v, type(v)
[] <type 'list'>
```

ellbil(*z0*, *z1*, *precision=0*)

e.ellbil(z0, z1): return the value of the canonical bilinear form on z0 and z1.

INPUT:

- `e` - elliptic curve (assumed integral given by a minimal model, as returned by ellminimalmodel)
- `z0, z1` - rational points on e

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit().ellminimalmodel()[0]
sage: e.ellbil([1, 0], [-1, 1])
0.418188984498861
```

ellchangecurve(*ch*)

e.ellchangecurve(ch): return the new model (equation) for the elliptic curve e given by the change of coordinates ch.

The change of coordinates is specified by a vector ch=[u,r,s,t]; if x' and y' are the new coordinates, then $x = u^2x' + r$ and $y = u^3y' + su^2x' + t$.

INPUT:

- `e` - elliptic curve
- `ch` - change of coordinates vector with 4 entries

EXAMPLES:

```
sage: e = pari([1,2,3,4,5]).ellinit()
sage: e.ellglobalred()
[10351, [1, -1, 0, -1], 1]
sage: f = e.ellchangecurve([1,-1,0,-1])
sage: f[:5]
[1, -1, 0, 4, 3]
```

ellchangepoint (*y*)

self.ellchangepoint(y): change data on point or vector of points self on an elliptic curve according to y=[u,r,s,t]

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit()
sage: x = pari([1,0])
sage: e.ellisoncurve([1,4])
False
sage: e.ellisoncurve(x)
True
sage: f = e.ellchangecurve([1,2,3,-1])
sage: f[:5]    # show only first five entries
[6, -2, -1, 17, 8]
sage: x.ellchangepoint([1,2,3,-1])
[-1, 4]
sage: f.ellisoncurve([-1,4])
True
```

elleisnum (*k*, *flag=0*, *precision=0*)

om.elleisnum(k, flag=0): om=[om1,om2] being a 2-component vector giving a basis of a lattice L and k an even positive integer, computes the numerical value of the Eisenstein series of weight k. When flag is non-zero and k=4 or 6, this gives g2 or g3 with the correct normalization.

INPUT:

- `om` - gen, 2-component vector giving a basis of a lattice L
- `k` - int (even positive)
- `flag` - int (default 0)

OUTPUT:

- `gen` - numerical value of E_k

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit()
sage: om = e.omega()
sage: om
[2.49021256085506, -1.97173770155165*I]
sage: om.elleisnum(2) # was:  -5.28864933965426
10.0672605281120
sage: om.elleisnum(4)
112.000000000000
sage: om.elleisnum(100)
2.15314248576078 E50
```

elleta (*precision=0*)

e.elleta(): return the vector [eta1,eta2] of quasi-periods associated with the period lattice e.omega() of the elliptic curve e.

EXAMPLES:

```
sage: e = pari([0,0,0,-82,0]).ellinit()
sage: e.elleta()
[3.60546360143265, 3.60546360143265*I]
sage: w1,w2 = e.omega()
sage: eta1, eta2 = e.elleta()
sage: w1*eta2-w2*eta1
6.28318530717959*I
```

```
sage: w1*eta2-w2*eta1 == pari(2*pi*I)
True
sage: pari([0,0,0,-82,0]).ellinit(flag=1).elleta()
Traceback (most recent call last):
...
PariError: incorrect type in elleta
```

ellglobalred()

e.ellglobalred(): return information related to the global minimal model of the elliptic curve e.

INPUT:

- e - elliptic curve (returned by ellinit)

OUTPUT:

- gen - the (arithmetic) conductor of e
- gen - a vector giving the coordinate change over Q from e to its minimal integral model (see also ellminimalmodel)
- gen - the product of the local Tamagawa numbers of e

EXAMPLES:

```
sage: e = pari([0, 5, 2, -1, 1]).ellinit()
sage: e.ellglobalred()
[20144, [1, -2, 0, -1], 1]
sage: e = pari(EllipticCurve('17a').a_invariants()).ellinit()
sage: e.ellglobalred()
[17, [1, 0, 0, 0], 4]
```

ellheight(*a*, *flag=2*, *precision=0*)

e.ellheight(a, flag=2): return the global Neron-Tate height of the point a on the elliptic curve e.

INPUT:

- e - elliptic curve over **Q**, assumed to be in a standard minimal integral model (as given by ellminimalmodel)
- a - rational point on e
- flag (optional) - specifies which algorithm to be used for computing the archimedean local height:
  - 0 - **uses sigma- and theta-functions and a trick** due to J. Silverman
  - 1 - uses Tate's 4^n algorithm
  - 2 - uses Mestre's AGM algorithm (this is the default, being faster than the other two)
- precision (optional) - the precision of the result, in bits.

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit().ellminimalmodel()[0]
sage: e.ellheight([1,0])
0.476711659343740
sage: e.ellheight([1,0], flag=0)
0.476711659343740
sage: e.ellheight([1,0], flag=1)
0.476711659343740
sage: e.ellheight([1,0], precision=128).sage()
0.47671165934373953737948605888465305932
```

ellheightmatrix (*x*, *precision=0*)

e.ellheightmatrix(x): return the height matrix for the vector x of points on the elliptic curve e.

In other words, it returns the Gram matrix of x with respect to the height bilinear form on e (see ellbil).

INPUT:

- `e` - elliptic curve over $\mathbf{Q}$, assumed to be in a standard minimal integral model (as given by ellminimalmodel)
- `x` - vector of rational points on e

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit().ellminimalmodel()[0]
sage: e.ellheightmatrix([[1,0], [-1,1]])
[0.476711659343740, 0.418188984498861; 0.418188984498861, 0.686667083305587]
```

It is allowed to call `ellinit()` with `flag=1`:

```
sage: E = pari([0,1,1,-2,0]).ellinit(flag=1)
sage: E.ellheightmatrix([[1,0], [-1,1]], precision=128).sage()
[0.47671165934373953737948605888465305932 0.41818898449886058562988945821587638244]
[0.41818898449886058562988945821587638244 0.68666708330558658572355210295409678904]
```

ellinit (*flag=0*, *precision=0*)

Return the PARI elliptic curve object with Weierstrass coefficients given by self, a list with 5 elements.

INPUT:

- `self` - a list of 5 coefficients
- `flag (optional, default: 0)` - if 0, ask for a PARI ell structure with 19 components; if 1, ask for a shorted PARI sell structure with only the first 13 components.
- `precision (optional, default: 0)` - the real precision to be used in the computation of the components of the PARI (s)ell structure; if 0, use the default 64 bits.

Note: The parameter `precision` in `ellinit` controls not only the real precision of the resulting (s)ell structure, but in some cases also the precision of most subsequent computations with this elliptic curve (if those rely on the precomputations done by `ellinit`). You should therefore set the precision from the start to the value you require.

OUTPUT:

- `gen` - either a PARI ell structure with 19 components (if flag=0), or a PARI sell structure with 13 components (if flag=1).

EXAMPLES: An elliptic curve with integer coefficients:

```
sage: e = pari([0,1,0,1,0]).ellinit(); e
[0, 1, 0, 1, 0, 4, 2, 0, -1, -32, 224, -48, 2048/3, [0.E-28, -0.500000000000000 - 0.86602540
[0, 1, 0, 1, 0, 4, 2, 0, -1, -32, 224, -48, 2048/3, [0.E-38, -0.500000000000000 - 0.86602540
```

Its inexact components have the default precision of 53 bits:

```
sage: RR(e[14])
3.37150070962519
```

We can compute this to higher precision:

```
sage: R = RealField(150)
sage: e = pari([0,1,0,1,0]).ellinit(precision=150)
```

```
sage: R(e[14])
3.37150070962519208574240731559815397900160l8
```

Using flag=1 returns a short elliptic curve PARI object:

```
sage: pari([0,1,0,1,0]).ellinit(flag=1)
[0, 1, 0, 1, 0, 4, 2, 0, -1, -32, 224, -48, 2048/3]
```

The coefficients can be any ring elements that convert to PARI:

```
sage: pari([0,1/2,0,-3/4,0]).ellinit(flag=1)
[0, 1/2, 0, -3/4, 0, 2, -3/2, 0, -9/16, 40, -116, 117/4, 256000/117]
sage: pari([0,0.5,0,-0.75,0]).ellinit(flag=1)
[0, 0.500000000000000, 0, -0.750000000000000, 0, 2.00000000000000, -1.50000000000000, 0, -0.
sage: pari([0,I,0,1,0]).ellinit(flag=1)
[0, I, 0, 1, 0, 4*I, 2, 0, -1, -64, 352*I, -80, 16384/5]
sage: pari([0,x,0,2*x,1]).ellinit(flag=1)
[0, x, 0, 2*x, 1, 4*x, 4*x, 4, -4*x^2 + 4*x, 16*x^2 - 96*x, -64*x^3 + 576*x^2 - 864, 64*x^4
```

ellisoncurve (*x*)

e.ellisoncurve(x): return True if the point x is on the elliptic curve e, False otherwise.

If the point or the curve have inexact coefficients, an attempt is made to take this into account.

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit()
sage: e.ellisoncurve([1,0])
True
sage: e.ellisoncurve([1,1])
False
sage: e.ellisoncurve([1,0.00000000000000001])
False
sage: e.ellisoncurve([1,0.000000000000000001])
True
sage: e.ellisoncurve([0])
True
```

ellj (*precision=0*)

Elliptic j-invariant of `self`.

EXAMPLES:

```
sage: pari(I).ellj()
1728.00000000000
sage: pari(3*I).ellj()
153553679.396729
sage: pari('quadgen(-3)').ellj()
0.E-54
sage: pari('quadgen(-7)').ellj(precision=256).sage()
-3375.000000000000000000000000000000000000000000000000000000000000000000000000
sage: pari(-I).ellj()
Traceback (most recent call last):
...
PariError: argument '-I' does not belong to upper half-plane
```

elllocalred (*p*)

e.elllocalred(p): computes the data of local reduction at the prime p on the elliptic curve e

For more details on local reduction and Kodaira types, see IV.8 and IV.9 in J. Silverman's book "Advanced topics in the arithmetic of elliptic curves".

INPUT:

- `e` - elliptic curve with coefficients in $\mathbf{Z}$
- `p` - prime number

OUTPUT:

- `gen` - the exponent of p in the arithmetic conductor of e
- `gen` - the Kodaira type of e at p, encoded as an integer:
- `1` - type I_0: good reduction, nonsingular curve of genus 1
- `2` - type II: rational curve with a cusp
- `3` - type III: two nonsingular rational curves intersecting tangentially at one point
- `4` - type IV: three nonsingular rational curves intersecting at one point
- `5` - type I_1: rational curve with a node
- `6 or larger` - think of it as $4+v$, then it is type I_v: v nonsingular rational curves arranged as a v-gon
- `-1` - type I_0^*: nonsingular rational curve of multiplicity two with four nonsingular rational curves of multiplicity one attached
- `-2` - type II^*: nine nonsingular rational curves in a special configuration
- `-3` - type III^*: eight nonsingular rational curves in a special configuration
- `-4` - type IV^*: seven nonsingular rational curves in a special configuration
- `-5 or smaller` - think of it as $-4-v$, then it is type I_v^*: chain of $v+1$ nonsingular rational curves of multiplicity two, with two nonsingular rational curves of multiplicity one attached at either end
- `gen` - a vector with 4 components, giving the coordinate changes done during the local reduction; if the first component is 1, then the equation for e was already minimal at p
- `gen` - the local Tamagawa number c_p

EXAMPLES:

Type I_0:

```
sage: e = pari([0,0,0,0,1]).ellinit()
sage: e.elllocalred(7)
[0, 1, [1, 0, 0, 0], 1]
```

Type II:

```
sage: e = pari(EllipticCurve('27a3').a_invariants()).ellinit()
sage: e.elllocalred(3)
[3, 2, [1, -1, 0, 1], 1]
```

Type III:

```
sage: e = pari(EllipticCurve('24a4').a_invariants()).ellinit()
sage: e.elllocalred(2)
[3, 3, [1, 1, 0, 1], 2]
```

Type IV:

```
sage: e = pari(EllipticCurve('20a2').a_invariants()).ellinit()
sage: e.elllocalred(2)
[2, 4, [1, 1, 0, 1], 3]
```

Type I_1:

```
sage: e = pari(EllipticCurve('11a2').a_invariants()).ellinit()
sage: e.elllocalred(11)
[1, 5, [1, 0, 0, 0], 1]
```

Type I_2:

```
sage: e = pari(EllipticCurve('14a4').a_invariants()).ellinit()
sage: e.elllocalred(2)
[1, 6, [1, 0, 0, 0], 2]
```

Type I_6:

```
sage: e = pari(EllipticCurve('14a1').a_invariants()).ellinit()
sage: e.elllocalred(2)
[1, 10, [1, 0, 0, 0], 2]
```

Type I_0^*:

```
sage: e = pari(EllipticCurve('32a3').a_invariants()).ellinit()
sage: e.elllocalred(2)
[5, -1, [1, 1, 1, 0], 1]
```

Type II^*:

```
sage: e = pari(EllipticCurve('24a5').a_invariants()).ellinit()
sage: e.elllocalred(2)
[3, -2, [1, 2, 1, 4], 1]
```

Type III^*:

```
sage: e = pari(EllipticCurve('24a2').a_invariants()).ellinit()
sage: e.elllocalred(2)
[3, -3, [1, 2, 1, 4], 2]
```

Type IV^*:

```
sage: e = pari(EllipticCurve('20a1').a_invariants()).ellinit()
sage: e.elllocalred(2)
[2, -4, [1, 0, 1, 2], 3]
```

Type I_1^*:

```
sage: e = pari(EllipticCurve('24a1').a_invariants()).ellinit()
sage: e.elllocalred(2)
[3, -5, [1, 0, 1, 2], 4]
```

Type I_6^*:

```
sage: e = pari(EllipticCurve('90c2').a_invariants()).ellinit()
sage: e.elllocalred(3)
[2, -10, [1, 96, 1, 316], 4]
```

ellseries(*s*, *A=1*, *precision=0*)

e.elllseries(s, A=1): return the value of the L-series of the elliptic curve e at the complex number s.

This uses an $O(N^{1/2})$ algorithm in the conductor N of e, so it is impractical for large conductors (say greater than 10^{12}).

INPUT:

- `e` - elliptic curve defined over **Q**
- `s` - complex number
- `A (optional)` - cutoff point for the integral, which must be chosen close to 1 for best speed.

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit()
sage: e.elllseries(2.1)
0.402838047956645
sage: e.elllseries(1, precision=128)
2.87490929644255 E-38
sage: e.elllseries(1, precision=256)
3.00282377034977 E-77
sage: e.elllseries(-2)
0
sage: e.elllseries(2.1, A=1.1)
0.402838047956645
```

ellminimalmodel()

ellminimalmodel(e): return the standard minimal integral model of the rational elliptic curve e and the corresponding change of variables. INPUT:

- `e` - gen (that defines an elliptic curve)

OUTPUT:

- `gen` - minimal model
- `gen` - change of coordinates

EXAMPLES:

```
sage: e = pari([1,2,3,4,5]).ellinit()
sage: F, ch = e.ellminimalmodel()
sage: F[:5]
[1, -1, 0, 4, 3]
sage: ch
[1, -1, 0, -1]
sage: e.ellchangecurve(ch)[:5]
[1, -1, 0, 4, 3]
```

ellorder(*x*)

e.ellorder(x): return the order of the point x on the elliptic curve e (return 0 if x is not a torsion point)

INPUT:

- `e` - elliptic curve defined over **Q**
- `x` - point on e

EXAMPLES:

```
sage: e = pari(EllipticCurve('65a1').a_invariants()).ellinit()
```

A point of order two:

```
sage: e.ellorder([0,0])
2
```

And a point of infinite order:

```
sage: e.ellorder([1,0])
0
```

ellordinate (*x*, *precision=0*)

e.ellordinate(x): return the y-coordinates of the points on the elliptic curve e having x as x-coordinate.

INPUT:

- e - elliptic curve
- x - x-coordinate (can be a complex or p-adic number, or a more complicated object like a power series)

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit()
sage: e.ellordinate(0)
[0, -1]
sage: e.ellordinate(I)
[0.582203589721741 - 1.38606082464177*I, -1.58220358972174 + 1.38606082464177*I]
sage: e.ellordinate(I, precision=128)[0].sage()
0.58220358972174117723338947874993600727 - 1.3860608246417697185311834209833653345*I
sage: e.ellordinate(1+3*5^1+O(5^3))
[4*5 + 5^2 + O(5^3), 4 + 3*5^2 + O(5^3)]
sage: e.ellordinate('z+2*z^2+O(z^4)')
[-2*z - 7*z^2 - 23*z^3 + O(z^4), -1 + 2*z + 7*z^2 + 23*z^3 + O(z^4)]
```

The field in which PARI looks for the point depends on the input field:

```
sage: e.ellordinate(5)
[]
sage: e.ellordinate(5.0)
[11.3427192823270, -12.3427192823270]
sage: e.ellordinate(RR(-3))
[-1/2 + 3.42782730020052*I, -1/2 - 3.42782730020052*I]
```

ellpointtoz (*pt*, *precision=0*)

e.ellpointtoz(pt): return the complex number (in the fundamental parallelogram) corresponding to the point `pt` on the elliptic curve e, under the complex uniformization of e given by the Weierstrass p-function.

The complex number z returned by this function lies in the parallelogram formed by the real and complex periods of e, as given by e.omega().

EXAMPLES:

```
sage: e = pari([0,0,0,1,0]).ellinit()
sage: e.ellpointtoz([0,0])
1.85407467730137
```

The point at infinity is sent to the complex number 0:

```
sage: e.ellpointtoz([0])
0
```

ellpow (*z*, *n*)

e.ellpow(z, n): return n times the point z on the elliptic curve e.

INPUT:

- e - elliptic curve
- z - point on e

- n - integer, or a complex quadratic integer of complex multiplication for e. Complex multiplication currently only works if e is defined over Q.

EXAMPLES: We consider a curve with CM by $Z[i]$:

```
sage: e = pari([0,0,0,3,0]).ellinit()
sage: p = [1,2]  # Point of infinite order
```

Multiplication by two:

```
sage: e.ellpow([0,0], 2)
[0]
sage: e.ellpow(p, 2)
[1/4, -7/8]
```

Complex multiplication:

```
sage: q = e.ellpow(p, 1+I); q
[-2*I, 1 + I]
sage: e.ellpow(q, 1-I)
[1/4, -7/8]
```

TESTS:

```
sage: for D in [-7, -8, -11, -12, -16, -19, -27, -28]:  # long time (1s)
....:     hcpol = hilbert_class_polynomial(D)
....:     j = hcpol.roots(multiplicities=False)[0]
....:     t = (1728-j)/(27*j)
....:     E = EllipticCurve([4*t,16*t^2])
....:     P = E.point([0, 4*t])
....:     assert(E.j_invariant() == j)
....:     #
....:     # Compute some CM number and its minimal polynomial
....:     #
....:     cm = pari('cm = (3*quadgen(%s)+2)'%D)
....:     cm_minpoly = pari('minpoly(cm)')
....:     #
....:     # Evaluate cm_minpoly(cm)(P), which should be zero
....:     #
....:     e = pari(E)  # Convert E to PARI
....:     P2 = e.ellpow(P, cm_minpoly[2]*cm + cm_minpoly[1])
....:     P0 = e.elladd(e.ellpow(P, cm_minpoly[0]), e.ellpow(P2, cm))
....:     assert(P0 == E(0))
```

ellrootno(*p=1*)

e.ellrootno(p): return the (local or global) root number of the L-series of the elliptic curve e

If p is a prime number, the local root number at p is returned. If p is 1, the global root number is returned. Note that the global root number is the sign of the functional equation of the L-series, and therefore conjecturally equal to the parity of the rank of e.

INPUT:

- e - elliptic curve over **Q**
- p (default = 1) - 1 or a prime number

OUTPUT: 1 or -1

EXAMPLES: Here is a curve of rank 3:

```
sage: e = pari([0,0,0,-82,0]).ellinit()
sage: e.ellrootno()
```

```
-1
sage: e.ellrootno(2)
1
sage: e.ellrootno(1009)
1
```

ellsigma (*z*, *flag=0*, *precision=0*)

e.ellsigma(z, flag=0): return the value at the complex point z of the Weierstrass σ function associated to the elliptic curve e.

EXAMPLES:

```
sage: e = pari([0,0,0,1,0]).ellinit()
sage: C.<i> = ComplexField()
sage: e.ellsigma(2+i)
1.43490215804166 + 1.80307856719256*I
```

ellsub (*z0*, *z1*)

e.ellsub(z0, z1): return z0-z1 on this elliptic curve.

INPUT:

- `e` - elliptic curve E
- `z0` - point on E
- `z1` - point on E

OUTPUT: point on E

EXAMPLES:

```
sage: e = pari([0, 1, 1, -2, 0]).ellinit()
sage: e.ellsub([1,0], [-1,1])
[0, 0]
```

elltaniyama ()

elltors (*flag=0*)

e.elltors(flag = 0): return information about the torsion subgroup of the elliptic curve e

INPUT:

- `e` - elliptic curve over **Q**
- `flag (optional)` - specify which algorithm to use:
- `0 (default)` - use Doud's algorithm: bound torsion by computing the cardinality of e(GF(p)) for small primes of good reduction, then look for torsion points using Weierstrass parametrization and Mazur's classification
- `1` - use algorithm given by the Nagell-Lutz theorem (this is much slower)

OUTPUT:

- `gen` - the order of the torsion subgroup, a.k.a. the number of points of finite order
- `gen` - vector giving the structure of the torsion subgroup as a product of cyclic groups, sorted in non-increasing order
- `gen` - vector giving points on e generating these cyclic groups

EXAMPLES:

```
sage: e = pari([1,0,1,-19,26]).ellinit()
sage: e.elltors()
[12, [6, 2], [[-2, 8], [3, -2]]]
```

ellwp(*z='z', n=20, flag=0, precision=0*)

Return the value or the series expansion of the Weierstrass P-function at z on the lattice $self$ (or the lattice defined by the elliptic curve $self$).

INPUT:

- `self` – an elliptic curve created using `ellinit` or a list `[om1, om2]` representing generators for a lattice.
- `z` – (default: 'z') a complex number or a variable name (as string or PARI variable).
- `n` – (default: 20) if 'z' is a variable, compute the series expansion up to at least $O(z^n)$.
- `flag` – (default = 0): If `flag` is 0, compute only $P(z)$. If `flag` is 1, compute $[P(z), P'(z)]$.

OUTPUT:

- $P(z)$ **(if `flag` is 0) or** $[P(z), P'(z)]$ **(if `flag` is 1).** numbers

EXAMPLES:

We first define the elliptic curve X_0(11):

```
sage: E = pari([0,-1,1,-10,-20]).ellinit()
```

Compute P(1):

```
sage: E.ellwp(1)
13.9658695257485 + 0.E-18*I
```

Compute P(1+i), where i = sqrt(-1):

```
sage: C.<i> = ComplexField()
sage: E.ellwp(pari(1+i))
-1.11510682565555 + 2.33419052307470*I
sage: E.ellwp(1+i)
-1.11510682565555 + 2.33419052307470*I
```

The series expansion, to the default $O(z^20)$ precision:

```
sage: E.ellwp()
z^-2 + 31/15*z^2 + 2501/756*z^4 + 961/675*z^6 + 77531/41580*z^8 + 1202285717/928746000*z^10
```

Compute the series for wp to lower precision:

```
sage: E.ellwp(n=4)
z^-2 + 31/15*z^2 + O(z^4)
```

Next we use the version where the input is generators for a lattice:

```
sage: pari([1.2692, 0.63 + 1.45*i]).ellwp(1)
13.9656146936689 + 0.000644829272810...*I
```

With flag=1, compute the pair P(z) and P'(z):

```
sage: E.ellwp(1, flag=1)
[13.9658695257485 + 0.E-18*I, 50.5619300880073 ... E-18*I]
```

ellzeta (*z*, *precision=0*)

e.ellzeta(z): return the value at the complex point z of the Weierstrass ζ function associated with the elliptic curve e.

Note: This function has infinitely many poles (one of which is at z=0); attempting to evaluate it too close to one of the poles will result in a PariError.

INPUT:

- e - elliptic curve
- z - complex number

EXAMPLES:

```
sage: e = pari([0,0,0,1,0]).ellinit()
sage: e.ellzeta(1)
1.06479841295883 + 0.E-19*I                 # 32-bit
1.06479841295883 + 5.42101086242752 E-20*I # 64-bit
sage: C.<i> = ComplexField()
sage: e.ellzeta(i-1)
-0.350122658523049 - 0.350122658523049*I
```

ellztopoint (*z*, *precision=0*)

e.ellztopoint(z): return the point on the elliptic curve e corresponding to the complex number z, under the usual complex uniformization of e by the Weierstrass p-function.

INPUT:

- e - elliptic curve
- z - complex number

OUTPUT point on e

EXAMPLES:

```
sage: e = pari([0,0,0,1,0]).ellinit()
sage: C.<i> = ComplexField()
sage: e.ellztopoint(1+i)
[0.E-19                - 1.02152286795670*I, -0.149072813701096 - 0.149072813701096*I]  # 32
[7.96075508054992 E-21 - 1.02152286795670*I, -0.149072813701096 - 0.149072813701096*I]  # 64
```

Complex numbers belonging to the period lattice of e are of course sent to the point at infinity on e:

```
sage: e.ellztopoint(0)
[0]
```

erfc (*x*, *precision=0*)

Return the complementary error function:

$$(2/\sqrt{\pi})\int_x^\infty e^{-t^2}dt.$$

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).erfc()
0.157299207050285
```

eta (*x*, *flag=0*, *precision=0*)

x.eta(flag=0): if flag=0, η function without the $q^{1/24}$; otherwise η of the complex number x in the upper half plane intelligently computed using $\mathrm{SL}(2, \mathbf{Z})$ transformations.

DETAILS: This functions computes the following. If the input x is a complex number with positive imaginary part, the result is $\prod_{n=1}^{\infty}(q - 1^n)$, where $q = e^{2i\pi x}$. If x is a power series (or can be converted to a power series) with positive valuation, the result is $\prod_{n=1}^{\infty}(1 - x^n)$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: C.<i> = ComplexField()
sage: pari(i).eta()
0.998129069925959
```

eval (**args*, ***kwds*)

Evaluate `self` with the given arguments.

This is currently implemented in 3 cases:

- univariate polynomials, rational functions, power series and Laurent series (using a single unnamed argument or keyword arguments),
- any PARI object supporting the PARI function `substvec` (in particular, multivariate polynomials) using keyword arguments,
- objects of type `t_CLOSURE` (functions in GP bytecode form) using unnamed arguments.

In no case is mixing unnamed and keyword arguments allowed.

EXAMPLES:

```
sage: f = pari('x^2 + 1')
sage: f.type()
't_POL'
sage: f.eval(I)
0
sage: f.eval(x=2)
5
sage: (1/f).eval(x=1)
1/2
```

The notation `f(x)` is an alternative for `f.eval(x)`:

```
sage: f(3) == f.eval(3)
True
```

Evaluating power series:

```
sage: f = pari('1 + x + x^3 + O(x^7)')
sage: f(2*pari('y')^2)
1 + 2*y^2 + 8*y^6 + O(y^14)
```

Substituting zero is sometimes possible, and trying to do so in illegal cases can raise various errors:

```
sage: pari('1 + O(x^3)').eval(0)
1
sage: pari('1/x').eval(0)
Traceback (most recent call last):
...
```

```
PariError: division by zero
sage: pari('1/x + O(x^2)').eval(0)
Traceback (most recent call last):
...
ZeroDivisionError: substituting 0 in Laurent series with negative valuation
sage: pari('1/x + O(x^2)').eval(pari('O(x^3)'))
Traceback (most recent call last):
...
PariError: division by zero
sage: pari('O(x^0)').eval(0)
Traceback (most recent call last):
...
PariError: non existent component in truecoeff
```

Evaluating multivariate polynomials:

```
sage: f = pari('y^2 + x^3')
sage: f(1)     # Dangerous, depends on PARI variable ordering
y^2 + 1
sage: f(x=1)   # Safe
y^2 + 1
sage: f(y=1)
x^3 + 1
sage: f(1, 2)
Traceback (most recent call last):
...
TypeError: evaluating PARI t_POL takes exactly 1 argument (2 given)
sage: f(y='x', x='2*y')
x^2 + 8*y^3
sage: f()
x^3 + y^2
```

It's not an error to substitute variables which do not appear:

```
sage: f.eval(z=37)
x^3 + y^2
sage: pari(42).eval(t=0)
42
```

We can define and evaluate closures as follows:

```
sage: T = pari('n -> n + 2')
sage: T.type()
't_CLOSURE'
sage: T.eval(3)
5

sage: T = pari('() -> 42')
sage: T()
42

sage: pr = pari('s -> print(s)')
sage: pr.eval('"hello world"')
hello world

sage: f = pari('myfunc(x,y) = x*y')
sage: f.eval(5, 6)
30
```

Default arguments work, missing arguments are treated as zero (like in GP):

```
sage: f = pari("(x, y, z=1.0) -> [x, y, z]")
sage: f(1, 2, 3)
[1, 2, 3]
sage: f(1, 2)
[1, 2, 1.00000000000000]
sage: f(1)
[1, 0, 1.00000000000000]
sage: f()
[0, 0, 1.00000000000000]
```

Using keyword arguments, we can substitute in more complicated objects, for example a number field:

```
sage: K.<a> = NumberField(x^2 + 1)
sage: nf = K._pari_()
sage: nf
[y^2 + 1, [0, 1], -4, 1, [Mat([1, 0.E-19 - 1.00000000000000*I]), [1, -1.00000000000000; 1, 1
sage: nf(y='x')
[x^2 + 1, [0, 1], -4, 1, [Mat([1, 0.E-19 - 1.00000000000000*I]), [1, -1.00000000000000; 1, 1
```

exp(*precision=0*)

x.exp(): exponential of x.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(0).exp()
1.00000000000000
sage: pari(1).exp()
2.71828182845905
sage: pari('x+O(x^8)').exp()
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + 1/720*x^6 + 1/5040*x^7 + O(x^8)
```

factor(*limit=-1*, *proof=1*)

Return the factorization of x.

INPUT:

- `limit` – (default: -1) is optional and can be set whenever x is of (possibly recursive) rational type. If limit is set return partial factorization, using primes up to limit (up to primelimit if limit=0).
- `proof` – (default: True) optional. If False (not the default), returned factors larger than 2^{64} may only be pseudoprimes.

Note: In the standard PARI/GP interpreter and C-library the factor command *always* has proof=False, so beware!

EXAMPLES:

```
sage: pari('x^10-1').factor()
[x - 1, 1; x + 1, 1; x^4 - x^3 + x^2 - x + 1, 1; x^4 + x^3 + x^2 + x + 1, 1]
sage: pari(2^100-1).factor()
[3, 1; 5, 3; 11, 1; 31, 1; 41, 1; 101, 1; 251, 1; 601, 1; 1801, 1; 4051, 1; 8101, 1; 268501,
sage: pari(2^100-1).factor(proof=False)
[3, 1; 5, 3; 11, 1; 31, 1; 41, 1; 101, 1; 251, 1; 601, 1; 1801, 1; 4051, 1; 8101, 1; 268501,
```

We illustrate setting a limit:

```
sage: pari(next_prime(10^50)*next_prime(10^60)*next_prime(10^4)).factor(10^5)
[10007, 1; 100000000000000000000000000000000000000000000000151000000000700000000000000000000
```

PARI doesn't have an algorithm for factoring multivariate polynomials:

```
sage: pari('x^3 - y^3').factor()
Traceback (most recent call last):
...
PariError: sorry, factor for general polynomials is not yet implemented
```

factormod(*p*, *flag=0*)

x.factormod(p,flag=0): factorization mod p of the polynomial x using Berlekamp. flag is optional, and can be 0: default or 1: simple factormod, same except that only the degrees of the irreducible factors are given.

factornf(*t*)

Factorization of the polynomial `self` over the number field defined by the polynomial `t`. This does not require that *t* is integral, nor that the discriminant of the number field can be factored.

EXAMPLES:

```
sage: x = polygen(QQ)
sage: K.<a> = NumberField(x^2 - 1/8)
sage: pari(x^2 - 2).factornf(K.pari_polynomial("a"))
[x + Mod(-4*a, 8*a^2 - 1), 1; x + Mod(4*a, 8*a^2 - 1), 1]
```

factorpadic(*p*, *r=20*, *flag=0*)

self.factorpadic(p,r=20,flag=0): p-adic factorization of the polynomial x to precision r. flag is optional and may be set to 0 (use round 4) or 1 (use Buchmann-Lenstra)

ffgen(*T*, *v=-1*)

Return the generator $g = x \bmod T$ of the finite field defined by the polynomial T.

INPUT:

- **T – a gen of type t_POL with coefficients of type t_INTMOD:** a polynomial over a prime finite field
- `v` – string: a variable name or -1 (optional)

If v is a string, then g will be a polynomial in v, else the variable of the polynomial T is used.

EXAMPLES:

```
sage: x = GF(2)['x'].gen()
sage: pari(x^2+x+2).ffgen()
x
sage: pari(x^2+x+1).ffgen('a')
a
```

ffinit(*p*, *n*, *v=-1*)

Return a monic irreducible polynomial g of degree n over the finite field of p elements.

INPUT:

- `p` – a gen of type t_INT: a prime number
- `n` – integer: the degree of the polynomial
- `v` – string: a variable name or -1 (optional)

If $v \geq 0', then`g$ will be a polynomial in v, else the variable x is used.

EXAMPLES:

```
sage: pari(7).ffinit(11)
Mod(1, 7)*x^11 + Mod(1, 7)*x^10 + Mod(4, 7)*x^9 + Mod(5, 7)*x^8 + Mod(1, 7)*x^7 + Mod(1, 7)*
sage: pari(2003).ffinit(3)
Mod(1, 2003)*x^3 + Mod(1, 2003)*x^2 + Mod(1993, 2003)*x + Mod(1995, 2003)
```

fibonacci(*x*)

Return the Fibonacci number of index x.

EXAMPLES:

```
sage: pari(18).fibonacci()
2584
sage: [pari(n).fibonacci() for n in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

floor(*x*)

For real x: return the largest integer = x. For rational functions: the quotient of numerator by denominator. For lists: apply componentwise.

INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari(5/9).floor()
0
sage: pari(11/9).floor()
1
sage: pari(1.17).floor()
1
sage: pari([1.5,2.3,4.99]).floor()
[1, 2, 4]
sage: pari([[1.1,2.2],[3.3,4.4]]).floor()
[[1, 2], [3, 4]]
sage: pari(x).floor()
x
sage: pari((x^2+x+1)/x).floor()
x + 1
sage: pari(x^2+5*x+2.5).floor()
x^2 + 5*x + 2.50000000000000

sage: pari('"hello world"').floor()
Traceback (most recent call last):
...
PariError: incorrect type in gfloor
```

frac(*x*)

frac(x): Return the fractional part of x, which is x - floor(x).

INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari(1.75).frac()
0.750000000000000
sage: pari(sqrt(2)).frac()
0.414213562373095
sage: pari('sqrt(-2)').frac()
Traceback (most recent call last):
...
PariError: incorrect type in gfloor
```

galoisapply(*aut, x*)

galoisfixedfield(*perm, flag=0, v=-1*)

galoisinit(*den=None*)

galoisinit(K{,den}): calculate Galois group of number field K; see PARI manual for meaning of den

galoispermtopol(*perm*)

gamma(*s, precision=0*)

s.gamma(precision): Gamma function at s.

If s is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If s is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).gamma()
1.00000000000000
sage: pari(5).gamma()
24.0000000000000
sage: C.<i> = ComplexField()
sage: pari(1+i).gamma()
0.498015668118356 - 0.154949828301811*I
```

TESTS:

```
sage: pari(-1).gamma()
Traceback (most recent call last):
...
PariError: non-positive integer argument in ggamma
```

gammah(*s, precision=0*)

s.gammah(): Gamma function evaluated at the argument x+1/2.

If s is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If s is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).gammah()
1.32934038817914
sage: pari(5).gammah()
52.3427777845535
sage: C.<i> = ComplexField()
sage: pari(1+i).gammah()
0.575315188063452 + 0.0882106775440939*I
```

gcd(*x, y=None*)

Return the greatest common divisor of x and y.

If y is None, then x must be a list or tuple, and the greatest common divisor of its components is returned.

EXAMPLES:

```
sage: pari(10).gcd(15)
5
sage: pari([5, 'y']).gcd()
1
sage: pari(['x', x^2]).gcd()
x
```

gequal(*a*, *b*)

Check whether a and b are equal using PARI's `gequal`.

EXAMPLES:

```
sage: a = pari(1); b = pari(1.0); c = pari('"some_string"')
sage: a.gequal(a)
True
sage: b.gequal(b)
True
sage: c.gequal(c)
True
sage: a.gequal(b)
True
sage: a.gequal(c)
False
```

WARNING: this relation is not transitive:

```
sage: a = pari('[0]'); b = pari(0); c = pari('[0,0]')
sage: a.gequal(b)
True
sage: b.gequal(c)
True
sage: a.gequal(c)
False
```

gequal0(*a*)

Check whether a is equal to zero.

EXAMPLES:

```
sage: pari(0).gequal0()
True
sage: pari(1).gequal0()
False
sage: pari(1e-100).gequal0()
False
sage: pari("0.0 + 0.0*I").gequal0()
True
sage: pari(GF(3^20,'t')(0)).gequal0()
True
```

gequal_long(*a*, *b*)

Check whether a is equal to the `long int` b using PARI's `gequalsg`.

EXAMPLES:

```
sage: a = pari(1); b = pari(2.0); c = pari('3*matid(3)')
sage: a.gequal_long(1)
True
```

```
sage: a.gequal_long(-1)
False
sage: a.gequal_long(0)
False
sage: b.gequal_long(2)
True
sage: b.gequal_long(-2)
False
sage: c.gequal_long(3)
True
sage: c.gequal_long(-3)
False
```

getattr (*attr*)

Return the PARI attribute with the given name.

EXAMPLES:

```
sage: K = pari("nfinit(x^2 - x - 1)")
sage: K.getattr("pol")
x^2 - x - 1
sage: K.getattr("disc")
5

sage: K.getattr("reg")
Traceback (most recent call last):
...
PariError: _.reg: incorrect type in reg
sage: K.getattr("zzz")
Traceback (most recent call last):
...
PariError: no function named "_.zzz"
```

hilbert (*x*, *y*, *p*)

hyperu (*a*, *b*, *x*, *precision=0*)

a.hyperu(b,x): U-confluent hypergeometric function.

If a, b, or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).hyperu(2,3)
0.333333333333333
```

idealadd (*x*, *y*)

idealaddtoone (*x*, *y*)

idealappr (*x*, *flag=0*)

idealcoprime (*x*, *y*)

Given two integral ideals x and y of a pari number field self, return an element a of the field (expressed in the integral basis of self) such that a*x is an integral ideal coprime to y.

EXAMPLES:

```
sage: F = NumberField(x^3-2, 'alpha')
sage: nf = F._pari_()
sage: x = pari('[1, -1, 2]~')
```

```
sage: y = pari('[1, -1, 3]~')
sage: nf.idealcoprime(x, y)
[1, 0, 0]~

sage: y = pari('[2, -2, 4]~')
sage: nf.idealcoprime(x, y)
[5/43, 9/43, -1/43]~
```

idealdiv(*x*, *y*, *flag=0*)

idealfactor(*x*)

idealhnf(*a*, *b=None*)

idealintersection(*x*, *y*)

ideallist(*bound*, *flag=4*)

Vector of vectors L of all idealstar of all ideals of $norm <= bound$.

The binary digits of flag mean:

- 1: give generators;
- 2: add units;
- 4: (default) give only the ideals and not the bid.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(x^2 + 1)
sage: L = K.pari_nf().ideallist(100)
```

Now we have our list L. Entry $L[n-1]$ contains all ideals of norm n:

```
sage: L[0]    # One ideal of norm 1.
[[1, 0; 0, 1]]
sage: L[64]   # 4 ideals of norm 65.
[[65, 8; 0, 1], [65, 47; 0, 1], [65, 18; 0, 1], [65, 57; 0, 1]]
```

ideallog(*x*, *bid*)

Return the discrete logarithm of the unit x in (ring of integers)/bid.

INPUT:

- `self` - a pari number field
- `bid` - a big ideal structure (corresponding to an ideal I of self) output by idealstar
- `x` - an element of self with valuation zero at all primes dividing I

OUTPUT:

- the discrete logarithm of x on the generators given in bid[2]

EXAMPLE:

```
sage: F = NumberField(x^3-2, 'alpha')
sage: nf = F._pari_()
sage: I = pari('[1, -1, 2]~')
sage: bid = nf.idealstar(I)
sage: x = pari('5')
sage: nf.ideallog(x, bid)
[25]~
```

idealmul (*x*, *y*, *flag=0*)

idealnorm (*x*)

idealprimedec (*nf*, *p*)

Prime ideal decomposition of the prime number p in the number field nf as a vector of 5 component vectors $[p, a, e, f, b]$ representing the prime ideals $pO_K + aO_K$, e ,'f' as usual, a as vector of components on the integral basis, b Lenstra's constant.

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: F = pari(K).idealprimedec(5); F
[[5, [-2, 1]~, 1, 1, [2, 1]~], [5, [2, 1]~, 1, 1, [-2, 1]~]]
sage: F[0].pr_get_p()
5
```

idealred (*I*, *vdir=0*)

idealstar (*I*, *flag=1*)

Return the big ideal (bid) structure of modulus I.

INPUT:

- `self` - a pari number field
- `I` – an ideal of self, or a row vector whose first component is an ideal and whose second component is a row vector of r_1 0 or 1.
- `flag` - determines the amount of computation and the shape of the output:
  - `1` (default): return a bid structure without generators
  - `2`: return a bid structure with generators (slower)
  - `0` (deprecated): only outputs units of (ring of integers/I) as an abelian group, i.e as a 3-component vector [h,d,g]: h is the order, d is the vector of SNF cyclic components and g the corresponding generators. This flag is deprecated: it is in fact slightly faster to compute a true bid structure, which contains much more information.

EXAMPLE:

```
sage: F = NumberField(x^3-2, 'alpha')
sage: nf = F._pari_()
sage: I = pari('[1, -1, 2]~')
sage: nf.idealstar(I)
[[[43, 9, 5; 0, 1, 0; 0, 0, 1], [0]], [42, [42]], Mat([[43, [9, 1, 0]~, 1, 1, [-5, -9, 1]~],
```

idealtwoelt (*x*, *a=None*)

idealval (*x*, *p*)

imag (*x*)

imag(x): Return the imaginary part of x. This function also works component-wise.

INPUT:

- `x` - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari('1+2*I').imag()
2
sage: pari(sqrt(-2)).imag()
```

```
1.41421356237310
sage: pari('x+I').imag()
1
sage: pari('x+2*I').imag()
2
sage: pari('(1+I)*x^2+2*I').imag()
x^2 + 2
sage: pari('[1,2,3] + [4*I,5,6]').imag()
[4, 0, 0]
```

incgam(*s*, *x*, *y=None*, *precision=0*)

s.incgam(x, y, precision): incomplete gamma function. y is optional and is the precomputed value of gamma(s).

If s is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If s is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: C.<i> = ComplexField()
sage: pari(1+i).incgam(3-i)
-0.0458297859919946 + 0.0433696818726677*I
```

incgamc(*s*, *x*, *precision=0*)

s.incgamc(x): complementary incomplete gamma function.

The arguments x and s are complex numbers such that s is not a pole of Γ and $|x|/(|s|+1)$ is not much larger than 1 (otherwise, the convergence is very slow). The function returns the value of the integral $\int_0^x e^{-t}t^{s-1}dt$.

If s or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).incgamc(2)
0.864664716763387
```

intformal(*y=-1*)

x.intformal(y): formal integration of x with respect to the main variable of y, or to the main variable of x if y is omitted

ispower(*k=None*)

Determine whether or not self is a perfect k-th power. If k is not specified, find the largest k so that self is a k-th power.

INPUT:

- `k` - int (optional)

OUTPUT:

- `power` - int, what power it is
- `g` - what it is a power of

EXAMPLES:

```
sage: pari(9).ispower()
(2, 3)
```

```
sage: pari(17).ispower()
(1, 17)
sage: pari(17).ispower(2)
(False, None)
sage: pari(17).ispower(1)
(1, 17)
sage: pari(2).ispower()
(1, 2)
```

isprime (*flag=0*)

isprime(x, flag=0): Returns True if x is a PROVEN prime number, and False otherwise.

INPUT:

- `flag` - int 0 (default): use a combination of algorithms. 1: certify primality using the Pocklington-Lehmer Test. 2: certify primality using the APRCL test.

OUTPUT:

- `bool` - True or False

EXAMPLES:

```
sage: pari(9).isprime()
False
sage: pari(17).isprime()
True
sage: n = pari(561)     # smallest Carmichael number
sage: n.isprime()       # not just a pseudo-primality test!
False
sage: n.isprime(1)
False
sage: n.isprime(2)
False
sage: n = pari(2^31-1)
sage: n.isprime(1)
(True, [2, 3, 1; 3, 5, 1; 7, 3, 1; 11, 3, 1; 31, 2, 1; 151, 3, 1; 331, 3, 1])
```

ispseudoprime (*flag=0*)

ispseudoprime(x, flag=0): Returns True if x is a pseudo-prime number, and False otherwise.

INPUT:

- `flag` - int 0 (default): checks whether x is a Baillie-Pomerance-Selfridge-Wagstaff pseudo prime (strong Rabin-Miller pseudo prime for base 2, followed by strong Lucas test for the sequence (P,-1), P smallest positive integer such that $P^2 - 4$ is not a square mod x). 0: checks whether x is a strong Miller-Rabin pseudo prime for flag randomly chosen bases (with end-matching to catch square roots of -1).

OUTPUT:

- `bool` - True or False, or when flag=1, either False or a tuple (True, cert) where `cert` is a primality certificate.

EXAMPLES:

```
sage: pari(9).ispseudoprime()
False
sage: pari(17).ispseudoprime()
True
sage: n = pari(561)      # smallest Carmichael number
```

```
sage: n.ispseudoprime(2)
False
```

issquare (*x*, *find_root=False*)

issquare(x,n): `True` if x is a square, `False` if not. If `find_root` is given, also returns the exact square root.

issquarefree ()

EXAMPLES:

```
sage: pari(10).issquarefree()
True
sage: pari(20).issquarefree()
False
```

j ()

e.j(): return the j-invariant of the elliptic curve e.

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.j()
-122023936/161051
sage: _.factor()
[-1, 1; 2, 12; 11, -5; 31, 3]
```

kronecker (*y*)

lcm (*x*, *y=None*)

Return the least common multiple of x and y.

If y is `None`, then x must be a list or tuple, and the least common multiple of its components is returned.

EXAMPLES:

```
sage: pari(10).lcm(15)
30
sage: pari([5, 'y']).lcm()
5*y
sage: pari([10, 'x', x^2]).lcm()
10*x^2
```

length ()

lex (*x*, *y*)

lex(x,y): Compare x and y lexicographically (1 if xy, 0 if x==y, -1 if xy)

lift (*x*, *v=-1*)

lift(x,v): Returns the lift of an element of Z/nZ to Z or R[x]/(P) to R[x] for a type R if v is omitted. If v is given, lift only polymods with main variable v. If v does not occur in x, lift only intmods.

INPUT:

- `x` - gen
- `v` - (optional) variable

OUTPUT: gen

EXAMPLES:

```
sage: x = pari("x")
sage: a = x.Mod('x^3 + 17*x + 3')
```

```
sage: a
Mod(x, x^3 + 17*x + 3)
sage: b = a^4; b
Mod(-17*x^2 - 3*x, x^3 + 17*x + 3)
sage: b.lift()
-17*x^2 - 3*x
```

??? more examples

lindep (*flag=0*)

list ()

Convert self to a list of PARI gens.

EXAMPLES:

A PARI vector becomes a Sage list:

```
sage: L = pari("vector(10,i,i^2)").list()
sage: L
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
sage: type(L)
<type 'list'>
sage: type(L[0])
<type 'sage.libs.pari.gen.gen'>
```

For polynomials, list() behaves as for ordinary Sage polynomials:

```
sage: pol = pari("x^3 + 5/3*x"); pol.list()
[0, 5/3, 0, 1]
```

For power series or Laurent series, we get all coefficients starting from the lowest degree term. This includes trailing zeros:

```
sage: R.<x> = LaurentSeriesRing(QQ)
sage: s = x^2 + O(x^8)
sage: s.list()
[1]
sage: pari(s).list()
[1, 0, 0, 0, 0, 0]
sage: s = x^-2 + O(x^0)
sage: s.list()
[1]
sage: pari(s).list()
[1, 0]
```

For matrices, we get a list of columns:

```
sage: M = matrix(ZZ,3,2,[1,4,2,5,3,6]); M
[1 4]
[2 5]
[3 6]
sage: pari(M).list()
[[1, 2, 3]~, [4, 5, 6]~]
```

For "scalar" types, we get a 1-element list containing `self`:

```
sage: pari("42").list()
[42]
```

list_str ()

Return str that might correctly evaluate to a Python-list.

listinsert (*obj*, *n*)

listput (*obj*, *n*)

lllgram ()

lllgramint ()

lngamma (*x*, *precision=0*)

This method is deprecated, please use `log_gamma()` instead.

See the `log_gamma()` method for documentation and examples.

EXAMPLES:

```
sage: pari(100).lngamma()
doctest:...: DeprecationWarning: The method lngamma() is deprecated. Use log_gamma() instea
See http://trac.sagemath.org/6992 for details.
359.134205369575
```

log (*x*, *precision=0*)

x.log(): natural logarithm of x.

This function returns the principal branch of the natural logarithm of x, i.e., the branch such that $\Im(\log(x)) \in]-\pi,\pi]$. The result is complex (with imaginary part equal to π) if $x \in \mathbf{R}$ and $x < 0$. In general, the algorithm uses the formula

$$\log(x) \simeq \frac{\pi}{2\text{agm}(1,4/s)} - m\log(2),$$

if $s = x2^m$ is large enough. (The result is exact to B bits provided that $s > 2^{B/2}$.) At low accuracies, this function computes $\log$ using the series expansion near 1.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

Note that p-adic arguments can also be given as input, with the convention that $\log(p) = 0$. Hence, in particular, $\exp(\log(x))/x$ is not in general equal to 1 but instead to a $(p-1)$-st root of unity (or ± 1 if $p = 2$) times a power of p.

EXAMPLES:

```
sage: pari(5).log()
1.60943791243410
sage: C.<i> = ComplexField()
sage: pari(i).log()
0.E-19 + 1.57079632679490*I
```

log_gamma (*x*, *precision=0*)

Logarithm of the gamma function of x.

This function returns the principal branch of the logarithm of the gamma function of x. The function $\log(\Gamma(x))$ is analytic on the complex plane with non-positive integers removed. This function can have much larger inputs than Γ itself.

The p-adic analogue of this function is unfortunately not implemented.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(100).log_gamma()
359.134205369575
```

matadjoint()

matadjoint(x): adjoint matrix of x.

EXAMPLES:

```
sage: pari('[1,2,3; 4,5,6;  7,8,9]').matadjoint()
[-3, 6, -3; 6, -12, 6; -3, 6, -3]
sage: pari('[a,b,c; d,e,f; g,h,i]').matadjoint()
[(i*e - h*f), (-i*b + h*c), (f*b - e*c); (-i*d + g*f), i*a - g*c, -f*a + d*c; (h*d - g*e), -
```

matdet(*flag=0*)

Return the determinant of this matrix.

INPUT:

- `flag` - (optional) flag 0: using Gauss-Bareiss. 1: use classical Gaussian elimination (slightly better for integer entries)

EXAMPLES:

```
sage: pari('[1,2; 3,4]').matdet(0)
-2
sage: pari('[1,2; 3,4]').matdet(1)
-2
```

matfrobenius(*flag=0*)

M.matfrobenius(flag=0): Return the Frobenius form of the square matrix M. If flag is 1, return only the elementary divisors (a list of polynomials). If flag is 2, return a two-components vector [F,B] where F is the Frobenius form and B is the basis change so that $M = B^{-1}FB$.

EXAMPLES:

```
sage: a = pari('[1,2;3,4]')
sage: a.matfrobenius()
[0, 2; 1, 5]
sage: a.matfrobenius(flag=1)
[x^2 - 5*x - 2]
sage: a.matfrobenius(2)
[[0, 2; 1, 5], [1, -1/3; 0, 1/3]]
sage: v = a.matfrobenius(2)
sage: v[0]
[0, 2; 1, 5]
sage: v[1]^(-1)*v[0]*v[1]
[1, 2; 3, 4]
```

We let t be the matrix of T_2 acting on modular symbols of level 43, which was computed using `ModularSymbols(43,sign=1).T(2).matrix()`:

```
sage: t = pari('[3, -2, 0, 0; 0, -2, 0, 1; 0, -1, -2, 2; 0, -2, 0, 2]')
sage: t.matfrobenius()
[0, 0, 0, -12; 1, 0, 0, -2; 0, 1, 0, 8; 0, 0, 1, 1]
sage: t.charpoly('x')
x^4 - x^3 - 8*x^2 + 2*x + 12
sage: t.matfrobenius(1)
[x^4 - x^3 - 8*x^2 + 2*x + 12]
```

AUTHORS:

- Martin Albrect (2006-04-02)

mathnf(*flag=0*)

A.mathnf(flag=0): (upper triangular) Hermite normal form of A, basis for the lattice formed by the columns of A.

INPUT:

- `flag` - optional, value range from 0 to 4 (0 if omitted), meaning : 0: naive algorithm
- `1: Use Batut's algorithm` - output 2-component vector [H,U] such that H is the HNF of A, and U is a unimodular matrix such that xU=H. 3: Use Batut's algorithm. Output [H,U,P] where P is a permutation matrix such that P A U = H. 4: As 1, using a heuristic variant of LLL reduction along the way.

EXAMPLES:

```
sage: pari('[1,2,3; 4,5,6;  7,8,9]').mathnf()
[6, 1; 3, 1; 0, 1]
```

mathnfmod(*d*)

Returns the Hermite normal form if d is a multiple of the determinant

Beware that PARI's concept of a Hermite normal form is an upper triangular matrix with the same column space as the input matrix.

INPUT:

- `d` - multiple of the determinant of self

EXAMPLES:

```
       sage: M=matrix([[1,2,3],[4,5,6],[7,8,11]])
sage: d=M.det()
sage: pari(M).mathnfmod(d)
       [6, 4, 3; 0, 1, 0; 0, 0, 1]
```

Note that d really needs to be a multiple of the discriminant, not just of the exponent of the cokernel:

```
       sage: M=matrix([[1,0,0],[0,2,0],[0,0,6]])
sage: pari(M).mathnfmod(6)
[1, 0, 0; 0, 1, 0; 0, 0, 6]
sage: pari(M).mathnfmod(12)
[1, 0, 0; 0, 2, 0; 0, 0, 6]
```

mathnfmodid(*d*)

Returns the Hermite Normal Form of M concatenated with d*Identity

Beware that PARI's concept of a Hermite normal form is a maximal rank upper triangular matrix with the same column space as the input matrix.

INPUT:

- `d` - Determines

EXAMPLES:

```
       sage: M=matrix([[1,0,0],[0,2,0],[0,0,6]])
sage: pari(M).mathnfmodid(6)
       [1, 0, 0; 0, 2, 0; 0, 0, 6]
```

This routine is not completely equivalent to mathnfmod:

```
sage: pari(M).mathnfmod(6)
[1, 0, 0; 0, 1, 0; 0, 0, 6]
```

matker (*flag=0*)

Return a basis of the kernel of this matrix.

INPUT:

- `flag` - optional; may be set to 0: default; non-zero: x is known to have integral entries.

EXAMPLES:

```
sage: pari('[1,2,3;4,5,6;7,8,9]').matker()
[1; -2; 1]
```

With algorithm 1, even if the matrix has integer entries the kernel need not be saturated (which is weird):

```
sage: pari('[1,2,3;4,5,6;7,8,9]').matker(1)
[3; -6; 3]
sage: pari('matrix(3,3,i,j,i)').matker()
[-1, -1; 1, 0; 0, 1]
sage: pari('[1,2,3;4,5,6;7,8,9]*Mod(1,2)').matker()
[Mod(1, 2); Mod(0, 2); Mod(1, 2)]
```

matkerint (*flag=0*)

Return the integer kernel of a matrix.

This is the LLL-reduced Z-basis of the kernel of the matrix x with integral entries.

INPUT:

- `flag` - optional, and may be set to 0: default, uses a modified LLL, 1: uses matrixqz.

EXAMPLES:

```
sage: pari('[2,1;2,1]').matker()
[-1/2; 1]
sage: pari('[2,1;2,1]').matkerint()
[1; -2]
sage: pari('[2,1;2,1]').matkerint(1)
[1; -2]
```

matsnf (*flag=0*)

x.matsnf(flag=0): Smith normal form (i.e. elementary divisors) of the matrix x, expressed as a vector d. Binary digits of flag mean 1: returns [u,v,d] where d=u*x*v, otherwise only the diagonal d is returned, 2: allow polynomial entries, otherwise assume x is integral, 4: removes all information corresponding to entries equal to 1 in d.

EXAMPLES:

```
sage: pari('[1,2,3; 4,5,6;  7,8,9]').matsnf()
[0, 3, 1]
```

matsolve (*B*)

matsolve(B): Solve the linear system Mx=B for an invertible matrix M

matsolve(B) uses Gaussian elimination to solve Mx=B, where M is invertible and B is a column vector.

The corresponding pari library routine is gauss. The gp-interface name matsolve has been given preference here.

INPUT:

•B - a column vector of the same dimension as the square matrix self

EXAMPLES:

```
sage: pari('[1,1;1,-1]').matsolve(pari('[1;0]'))
[1/2; 1/2]
```

matsolvemod(*D*, *B*, *flag=0*)

For column vectors $D = (d_i)$ and $B = (b_i)$, find a small integer solution to the system of linear congruences

$$R_i x = b_i \text{ (mod } d_i),$$

where R_i is the ith row of `self`. If $d_i = 0$, the equation is considered over the integers. The entries of `self`, D, and B should all be integers (those of D should also be non-negative).

If `flag` is 1, the output is a two-component row vector whose first component is a solution and whose second component is a matrix whose columns form a basis of the solution set of the homogeneous system.

For either value of `flag`, the output is 0 if there is no solution.

Note that if D or B is an integer, then it will be considered as a vector all of whose entries are that integer.

EXAMPLES:

```
sage: D = pari('[3,4]~')
sage: B = pari('[1,2]~')
sage: M = pari('[1,2;3,4]')
sage: M.matsolvemod(D, B)
[-2, 0]~
sage: M.matsolvemod(3, 1)
[-1, 1]~
sage: M.matsolvemod(pari('[3,0]~'), pari('[1,2]~'))
[6, -4]~
sage: M2 = pari('[1,10;9,18]')
sage: M2.matsolvemod(3, pari('[2,3]~'), 1)
[[0, -1]~, [-1, -2; 1, -1]]
sage: M2.matsolvemod(9, pari('[2,3]~'))
0
sage: M2.matsolvemod(9, pari('[2,45]~'), 1)
[[1, 1]~, [-1, -4; 1, -5]]
```

mattranspose()

Transpose of the matrix self.

EXAMPLES:

```
sage: pari('[1,2,3; 4,5,6; 7,8,9]').mattranspose()
[1, 4, 7; 2, 5, 8; 3, 6, 9]
```

max(*x*, *y*)

max(x,y): Return the maximum of x and y.

min(*x*, *y*)

min(x,y): Return the minimum of x and y.

mod()

Given an INTMOD or POLMOD `Mod(a,m)`, return the modulus m.

EXAMPLES:

```
sage: pari(4).Mod(5).mod()
5
```

```
sage: pari("Mod(x, x*y)").mod()
y*x
sage: pari("[Mod(4,5)]").mod()
Traceback (most recent call last):
...
TypeError: Not an INTMOD or POLMOD in mod()
```

modreverse()

modreverse(x): reverse polymod of the polymod x, if it exists.

EXAMPLES:

moebius(*x*)

moebius(x): Moebius function of x.

ncols()

Return the number of columns of self.

EXAMPLES:

```
sage: pari('matrix(19,8)').ncols()
8
```

newtonpoly(*p*)

x.newtonpoly(p): Newton polygon of polynomial x with respect to the prime p.

EXAMPLES:

```
sage: x = pari('y^8+6*y^6-27*y^5+1/9*y^2-y+1')
sage: x.newtonpoly(3)
[1, 1, -1/3, -1/3, -1/3, -1/3, -1/3, -1/3]
```

nextprime(*add_one=0*)

nextprime(x): smallest pseudoprime greater than or equal to x. If `add_one` is non-zero, return the smallest pseudoprime strictly greater than x.

EXAMPLES:

```
sage: pari(1).nextprime()
2
sage: pari(2).nextprime()
2
sage: pari(2).nextprime(add_one = 1)
3
sage: pari(2^100).nextprime()
1267650600228229401496703205653
```

nf_get_diff()

Returns the different of this number field as a PARI ideal.

INPUT:

- **`self` – A PARI number field being the output of `nfinit()`,** `bnfinit()` or `bnrinit()`.

EXAMPLES:

```
sage: K.<a> = NumberField(x^4 - 4*x^2 + 1)
sage: pari(K).nf_get_diff()
[12, 0, 0, 0; 0, 12, 8, 0; 0, 0, 4, 0; 0, 0, 0, 4]
```

nf_get_pol()

Returns the defining polynomial of this number field.

INPUT:

- **self – A PARI number field being the output of nfinit(),** bnfinit() or bnrinit().

EXAMPLES:

```
sage: K.<a> = NumberField(x^4 - 4*x^2 + 1)
sage: pari(K).nf_get_pol()
y^4 - 4*y^2 + 1
sage: bnr = pari("K = bnfinit(x^4 - 4*x^2 + 1); bnrinit(K, 2*x)")
sage: bnr.nf_get_pol()
x^4 - 4*x^2 + 1
```

For relative extensions, this returns the absolute polynomial, not the relative one:

```
sage: L.<b> = K.extension(x^2 - 5)
sage: pari(L).nf_get_pol()   # Absolute polynomial
y^8 - 28*y^6 + 208*y^4 - 408*y^2 + 36
sage: L.pari_rnf().nf_get_pol()
x^8 - 28*x^6 + 208*x^4 - 408*x^2 + 36
```

TESTS:

```
sage: x = polygen(QQ)
sage: K.<a> = NumberField(x^4 - 4*x^2 + 1)
sage: K.pari_nf().nf_get_pol()
y^4 - 4*y^2 + 1
sage: K.pari_bnf().nf_get_pol()
y^4 - 4*y^2 + 1
```

An error is raised for invalid input:

```
sage: pari("[0]").nf_get_pol()
Traceback (most recent call last):
...
PariError: incorrect type in pol
```

nf_get_sign()

Returns a Python list [r1, r2], where r1 and r2 are Python ints representing the number of real embeddings and pairs of complex embeddings of this number field, respectively.

INPUT:

- **self – A PARI number field being the output of nfinit(),** bnfinit() or bnrinit().

EXAMPLES:

```
sage: K.<a> = NumberField(x^4 - 4*x^2 + 1)
sage: s = K.pari_nf().nf_get_sign(); s
[4, 0]
sage: type(s); type(s[0])
<type 'list'>
<type 'int'>
sage: CyclotomicField(15).pari_nf().nf_get_sign()
[0, 4]
```

nf_get_zk()

Returns a vector with a $\mathbf{Z}$-basis for the ring of integers of this number field. The first element is always 1.

INPUT:

- **self – A PARI number field being the output of nfinit(),** bnfinit() or bnrinit().

EXAMPLES:

```
sage: K.<a> = NumberField(x^4 - 4*x^2 + 1)
sage: pari(K).nf_get_zk()
[1, y, y^3 - 4*y, y^2 - 2]
```

nf_subst (*z*)

Given a PARI number field `self`, return the same PARI number field but in the variable `z`.

INPUT:

- **`self` – A PARI number field being the output of `nfinit()`,** `bnfinit()` or `bnrinit()`.

EXAMPLES:

```
sage: x = polygen(QQ)
sage: K = NumberField(x^2 + 5, 'a')
```

We can substitute in a PARI `nf` structure:

```
sage: Kpari = K.pari_nf()
sage: Kpari.nf_get_pol()
y^2 + 5
sage: Lpari = Kpari.nf_subst('a')
sage: Lpari.nf_get_pol()
a^2 + 5
```

We can also substitute in a PARI `bnf` structure:

```
sage: Kpari = K.pari_bnf()
sage: Kpari.nf_get_pol()
y^2 + 5
sage: Kpari.bnf_get_cyc()   # Structure of class group
[2]
sage: Lpari = Kpari.nf_subst('a')
sage: Lpari.nf_get_pol()
a^2 + 5
sage: Lpari.bnf_get_cyc()   # We still have a bnf after substituting
[2]
```

nfbasis (*flag=0*, *fa=None*)

nfbasis(x, flag, fa): integral basis of the field QQ[a], where `a` is a root of the polynomial x.

Binary digits of `flag` mean:

- **1: assume that no square of a prime>primelimit divides the** discriminant of `x`.
- 2: use round 2 algorithm instead of round 4.

If present, `fa` provides the matrix of a partial factorization of the discriminant of `x`, useful if one wants only an order maximal at certain primes only.

EXAMPLES:

```
sage: pari('x^3 - 17').nfbasis()
[1, x, 1/3*x^2 - 1/3*x + 1/3]
```

We test `flag` = 1, noting it gives a wrong result when the discriminant (-4 * $p`^2 * `q$ in the example below) has a big square factor:

```
sage: p = next_prime(10^10); q = next_prime(p)
sage: x = polygen(QQ); f = x^2 + p^2*q
sage: pari(f).nfbasis(1)    # Wrong result
[1, x]
```

```
sage: pari(f).nfbasis()     # Correct result
[1, 1/10000000019*x]
sage: pari(f).nfbasis(fa = "[2,2; %s,2]"%p)     # Correct result and faster
[1, 1/10000000019*x]
```

TESTS:

`flag` = 2 should give the same result:

```
sage: pari('x^3 - 17').nfbasis(flag = 2)
[1, x, 1/3*x^2 - 1/3*x + 1/3]
```

nfbasis_d(*flag=0*, *fa=None*)

nfbasis_d(x): Return a basis of the number field defined over QQ by x and its discriminant.

EXAMPLES:

```
sage: F = NumberField(x^3-2,'alpha')
sage: F._pari_()[0].nfbasis_d()
([1, y, y^2], -108)

sage: G = NumberField(x^5-11,'beta')
sage: G._pari_()[0].nfbasis_d()
([1, y, y^2, y^3, y^4], 45753125)

sage: pari([-2,0,0,1]).Polrev().nfbasis_d()
([1, x, x^2], -108)
```

nfbasistoalg(*nf*, *x*)

Transforms the column vector `x` on the integral basis into an algebraic number.

INPUT:

- `nf` – a number field
- `x` – a column of rational numbers of length equal to the degree of `nf` or a single rational number

OUTPUT:

- A POLMOD representing the element of `nf` whose coordinates are `x` in the Z-basis of `nf`.

EXAMPLES:

```
sage: x = polygen(QQ)
sage: K.<a> = NumberField(x^3 - 17)
sage: Kpari = K.pari_nf()
sage: Kpari.getattr('zk')
[1, 1/3*y^2 - 1/3*y + 1/3, y]
sage: Kpari.nfbasistoalg(42)
Mod(42, y^3 - 17)
sage: Kpari.nfbasistoalg("[3/2, -5, 0]~")
Mod(-5/3*y^2 + 5/3*y - 1/6, y^3 - 17)
sage: Kpari.getattr('zk') * pari("[3/2, -5, 0]~")
-5/3*y^2 + 5/3*y - 1/6
```

nfbasistoalg_lift(*nf*, *x*)

Transforms the column vector `x` on the integral basis into a polynomial representing the algebraic number.

INPUT:

- `nf` – a number field
- `x` – a column of rational numbers of length equal to the degree of `nf` or a single rational number

OUTPUT:

- `nf.nfbasistoalg(x).lift()`

EXAMPLES:

```
sage: x = polygen(QQ)
sage: K.<a> = NumberField(x^3 - 17)
sage: Kpari = K.pari_nf()
sage: Kpari.getattr('zk')
[1, 1/3*y^2 - 1/3*y + 1/3, y]
sage: Kpari.nfbasistoalg_lift(42)
42
sage: Kpari.nfbasistoalg_lift("[3/2, -5, 0]~")
-5/3*y^2 + 5/3*y - 1/6
sage: Kpari.getattr('zk') * pari("[3/2, -5, 0]~")
-5/3*y^2 + 5/3*y - 1/6
```

nfdisc (*flag=0*, *p=0*)

nfdisc(x): Return the discriminant of the number field defined over QQ by x.

EXAMPLES:

```
sage: F = NumberField(x^3-2,'alpha')
sage: F._pari_()[0].nfdisc()
-108

sage: G = NumberField(x^5-11,'beta')
sage: G._pari_()[0].nfdisc()
45753125

sage: f = x^3-2
sage: f._pari_()
x^3 - 2
sage: f._pari_().nfdisc()
-108
```

nfeltdiveuc (*x*, *y*)

Given x and y in the number field `self`, return q such that $x - qy$ is "small".

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 5)
sage: x = 10
sage: y = a + 1
sage: pari(k).nfeltdiveuc(pari(x), pari(y))
[2, -2]~
```

nfeltreduce (*x*, *I*)

Given an ideal I in Hermite normal form and an element x of the pari number field self, finds an element r in self such that x-r belongs to the ideal and r is small.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 5)
sage: I = k.ideal(a)
sage: kp = pari(k)
sage: kp.nfeltreduce(12, I.pari_hnf())
[2, 0]~
sage: 12 - k(kp.nfeltreduce(12, I.pari_hnf())) in I
True
```

nffactor(*x*)

nfgaloisconj(*flag=0*, *denom=None*, *precision=0*)

Edited from the pari documentation:

nfgaloisconj(nf): list of conjugates of a root of the polynomial x=nf.pol in the same number field.

Uses a combination of Allombert's algorithm and nfroots.

EXAMPLES:

```
sage: x = QQ['x'].0; nf = pari(x^2 + 2).nfinit()
sage: nf.nfgaloisconj()
[-x, x]~
sage: nf = pari(x^3 + 2).nfinit()
sage: nf.nfgaloisconj()
[x]~
sage: nf = pari(x^4 + 2).nfinit()
sage: nf.nfgaloisconj()
[-x, x]~
```

nfgenerator()

nfhilbert(*a*, *b*, *p=None*)

nfhilbert(nf,a,b,{p}): if p is omitted, global Hilbert symbol (a,b) in nf, that is 1 if X^2-aY^2-bZ^2 has a non-trivial solution (X,Y,Z) in nf, -1 otherwise. Otherwise compute the local symbol modulo the prime ideal p.

EXAMPLES:

```
sage: x = polygen(QQ)
sage: K.<t> = NumberField(x^3 - x + 1)
sage: pari(K).nfhilbert(t, t + 2)
-1
sage: P = K.ideal(t^2 + t - 2)    # Prime ideal above 5
sage: pari(K).nfhilbert(t, t + 2, P.pari_prime())
-1
sage: P = K.ideal(t^2 + 3*t - 1) # Prime ideal above 23, ramified
sage: pari(K).nfhilbert(t, t + 2, P.pari_prime())
1
```

nfhnf(*x*)

nfhnf(nf,x) : given a pseudo-matrix (A, I) or an integral pseudo-matrix (A,I,J), finds a pseudo-basis in Hermite normal form of the module it generates.

A pseudo-matrix is a 2-component row vector (A, I) where A is a relative m x n matrix and I an ideal list of length n. An integral pseudo-matrix is a 3-component row vector (A, I, J).

Note: The definition of a pseudo-basis ([Cohen]): Let M be a finitely generated, torsion-free R-module, and set V = KM. If $\mathfrak{a}_i$ are fractional ideals of R and w_i are elements of V, we say that $(w_i, \mathfrak{a}_k)_{1\leq i\leq k}$ is a pseudo-basis of M if $M = \mathfrak{a}_1 w_1 \oplus \cdots \oplus \mathfrak{a}_k w_k$.

REFERENCES:

EXAMPLES:

```
sage: F.<a> = NumberField(x^2-x-1)
sage: Fp = pari(F)
sage: A = matrix(F,[[1,2,a,3],[3,0,a+2,0],[0,0,a,2],[3+a,a,0,1]])
sage: I = [F.ideal(-2*a+1),F.ideal(7), F.ideal(3),F.ideal(1)]
sage: Fp.nfhnf([pari(A),[pari(P) for P in I]])
```

```
[[1, [-969/5, -1/15]~, [15, -2]~, [-1938, -3]~; 0, 1, 0, 0; 0, 0, 1, 0;
0, 0, 0, 1], [[3997, 1911; 0, 7], [15, 6; 0, 3], [1, 0; 0, 1], [1, 0; 0,
1]]]
sage: K.<b> = NumberField(x^3-2)
sage: Kp = pari(K)
sage: A = matrix(K,[[1,0,0,5*b],[1,2*b^2,b,57],[0,2,1,b^2-3],[2,0,0,b]])
sage: I = [K.ideal(2),K.ideal(3+b^2),K.ideal(1),K.ideal(1)]
sage: Kp.nfhnf([pari(A),[pari(P) for P in I]])
[[1, -225, 72, -31; 0, 1, [0, -1, 0]~, [0, 0, -1/2]~; 0, 0, 1, [0, 0,
-1/2]~; 0, 0, 0, 1], [[1116, 756, 612; 0, 18, 0; 0, 0, 18], [2, 0, 0; 0,
2, 0; 0, 0, 2], [1, 0, 0; 0, 1, 0; 0, 0, 1], [2, 0, 0; 0, 1, 0; 0, 0,
1]]]
```

An example where the ring of integers of the number field is not a PID:

```
sage: K.<b> = NumberField(x^2+5)
sage: Kp = pari(K)
sage: A = matrix(K,[[1,0,0,5*b],[1,2*b^2,b,57],[0,2,1,b^2-3],[2,0,0,b]])
sage: I = [K.ideal(2),K.ideal(3+b^2),K.ideal(1),K.ideal(1)]
sage: Kp.nfhnf([pari(A),[pari(P) for P in I]])
[[1, [15, 6]~, [0, -54]~, [113, 72]~; 0, 1, [-4, -1]~, [0, -1]~; 0, 0,
1, 0; 0, 0, 0, 1], [[360, 180; 0, 180], [6, 4; 0, 2], [1, 0; 0, 1], [1,
0; 0, 1]]]
sage: A = matrix(K,[[1,0,0,5*b],[1,2*b,b,57],[0,2,1,b-3],[2,0,b,b]])
sage: I = [K.ideal(2).factor()[0][0],K.ideal(3+b),K.ideal(1),K.ideal(1)]
sage: Kp.nfhnf([pari(A),[pari(P) for P in I]])
[[1, [7605, 4]~, [5610, 5]~, [7913, -6]~; 0, 1, 0, -1; 0, 0, 1, 0; 0, 0,
0, 1], [[19320, 13720; 0, 56], [2, 1; 0, 1], [1, 0; 0, 1], [1, 0; 0,
1]]]
```

AUTHORS:

•Aly Deines (2012-09-19)

nfinit (*flag=0*, *precision=0*)

nfinit(pol, {flag=0}): `pol` being a nonconstant irreducible polynomial, gives a vector containing all the data necessary for PARI to compute in this number field.

`flag` is optional and can be set to:

- 0: default
- 1: do not compute different
- 2: first use polred to find a simpler polynomial
- **3: outputs a two-element vector [nf,Mod(a,P)], where nf is as in 2** and Mod(a,P) is a polmod equal to Mod(x,pol) and P=nf.pol

EXAMPLES:

```
sage: pari('x^3 - 17').nfinit()
[x^3 - 17, [1, 1], -867, 3, [[1, 1.68006..., 2.57128...; 1, -0.340034... + 2.65083...*I, -1.
```

TESTS:

This example only works after increasing precision:

```
sage: pari('x^2 + 10^100 + 1').nfinit(precision=64)
Traceback (most recent call last):
...
PariError: precision too low in floorr (precision loss in truncation)
```

```
sage: pari('x^2 + 10^100 + 1').nfinit()
[...]
```

Throw a PARI error which is not of type `precer`:

```
sage: pari('1.0').nfinit()
Traceback (most recent call last):
...
PariError: incorrect type in checknf
```

nfisisom(*other*)

nfisisom(x, y): Determine if the number fields defined by x and y are isomorphic. According to the PARI documentation, this is much faster if at least one of x or y is a number field. If they are isomorphic, it returns an embedding for the generators. If not, returns 0.

EXAMPLES:

```
sage: F = NumberField(x^3-2,'alpha')
sage: G = NumberField(x^3-2,'beta')
sage: F._pari_().nfisisom(G._pari_())
[y]

sage: GG = NumberField(x^3-4,'gamma')
sage: F._pari_().nfisisom(GG._pari_())
[1/2*y^2]

sage: F._pari_().nfisisom(GG.pari_nf())
[1/2*y^2]

sage: F.pari_nf().nfisisom(GG._pari_()[0])
[y^2]

sage: H = NumberField(x^2-2,'alpha')
sage: F._pari_().nfisisom(H._pari_())
0
```

nfroots(*poly*)

Return the roots of *poly* in the number field self without multiplicity.

EXAMPLES:

```
sage: y = QQ['yy'].0; _ = pari(y) # pari has variable ordering rules
sage: x = QQ['zz'].0; nf = pari(x^2 + 2).nfinit()
sage: nf.nfroots(y^2 + 2)
[Mod(-zz, zz^2 + 2), Mod(zz, zz^2 + 2)]
sage: nf = pari(x^3 + 2).nfinit()
sage: nf.nfroots(y^3 + 2)
[Mod(zz, zz^3 + 2)]
sage: nf = pari(x^4 + 2).nfinit()
sage: nf.nfroots(y^4 + 2)
[Mod(-zz, zz^4 + 2), Mod(zz, zz^4 + 2)]
```

nfrootsof1()

nf.nfrootsof1()

number of roots of unity and primitive root of unity in the number field nf.

EXAMPLES:

```
sage: nf = pari('x^2 + 1').nfinit()
sage: nf.nfrootsof1()
[4, -x]
```

nfsubfields(*d=0*)

Find all subfields of degree d of number field nf (all subfields if d is null or omitted). Result is a vector of subfields, each being given by [g,h], where g is an absolute equation and h expresses one of the roots of g in terms of the root x of the polynomial defining nf.

INPUT:

- `self` - nf number field
- `d` - C long integer

norm()

nrows()

Return the number of rows of self.

EXAMPLES:

```
sage: pari('matrix(19,8)').nrows()
19
```

numbpart(*x*)

numbpart(x): returns the number of partitions of x.

EXAMPLES:

```
sage: pari(20).numbpart()
627
sage: pari(100).numbpart()
190569292
```

numdiv(*n*)

Return the number of divisors of the integer n.

EXAMPLES:

```
sage: pari(10).numdiv()
4
```

numerator(*x*)

numerator(x): Returns the numerator of x.

INPUT:

- `x` - gen

OUTPUT: gen

EXAMPLES:

numtoperm(*k*, *n*)

numtoperm(k, n): Return the permutation number k (mod n!) of n letters, where n is an integer.

INPUT:

- `k` - gen, integer
- `n` - int

OUTPUT:

- gen - vector (permutation of 1,...,n)

EXAMPLES:

omega()

e.omega(): return basis for the period lattice of the elliptic curve e.

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.omega()
[1.26920930427955, -0.634604652139777 - 1.45881661693850*I]
```

order()

padicappr(*a*)

x.padicappr(a): p-adic roots of the polynomial x congruent to a mod p

padicprec(*x*, *p*)

padicprec(x,p): Return the absolute p-adic precision of the object x.

INPUT:

- x - gen

OUTPUT: int

EXAMPLES:

```
sage: K = Qp(11,5)
sage: x = K(11^-10 + 5*11^-7 + 11^-6)
sage: y = pari(x)
sage: y.padicprec(11)
-5
sage: y.padicprec(17)
Traceback (most recent call last):
...
PariError: not the same prime in padicprec
```

This works for polynomials too:

```
sage: R.<t> = PolynomialRing(Zp(3))
sage: pol = R([O(3^4), O(3^6), O(3^5)])
sage: pari(pol).padicprec(3)
4
```

padicprime(*x*)

The uniformizer of the p-adic ring this element lies in, as a t_INT.

INPUT:

- x - gen, of type t_PADIC

OUTPUT:

- p - gen, of type t_INT

EXAMPLES:

```
sage: K = Qp(11,5)
sage: x = K(11^-10 + 5*11^-7 + 11^-6)
sage: y = pari(x)
sage: y.padicprime()
11
```

```
sage: y.padicprime().type()
't_INT'
```

permtonum(*x*)

permtonum(x): Return the ordinal (between 1 and n!) of permutation vector x. ??? Huh ??? say more. what is a perm vector. 0 to n-1 or 1-n.

INPUT:

- x - gen (vector of integers)

OUTPUT:

- gen - integer

EXAMPLES:

phi(*n*)

Return the Euler phi function of n. EXAMPLES:

```
sage: pari(10).phi()
4
```

polcoeff(*n*, *var=-1*)

EXAMPLES:

```
sage: f = pari("x^2 + y^3 + x*y")
sage: f
x^2 + y*x + y^3
sage: f.polcoeff(1)
y
sage: f.polcoeff(3)
0
sage: f.polcoeff(3, "y")
1
sage: f.polcoeff(1, "y")
x
```

polcompositum(*pol2*, *flag=0*)

poldegree(*var=-1*)

f.poldegree(var=x): Return the degree of this polynomial.

poldisc(*var=-1*)

Return the discriminant of this polynomial.

EXAMPLES:

```
sage: pari("x^2 + 1").poldisc()
-4
```

Before trac ticket #15654, this used to take a very long time. Now it takes much less than a second:

```
sage: pari.allocatemem(200000)
PARI stack size set to 200000 bytes
sage: x = polygen(ZpFM(3,10))
sage: pol = ((x-1)^50 + x)
sage: pari(pol).poldisc()
2*3 + 3^4 + 2*3^6 + 3^7 + 2*3^8 + 2*3^9 + O(3^10)
```

poldiscreduced()

polgalois(*precision=0*)
f.polgalois(): Galois group of the polynomial f

polhensellift(*y, p, e*)
self.polhensellift(y, p, e): lift the factorization y of self modulo p to a factorization modulo p^e using Hensel lift. The factors in y must be pairwise relatively prime modulo p.

polinterpolate(*ya, x*)
self.polinterpolate(ya,x,e): polynomial interpolation at x according to data vectors self, ya (i.e. return P such that P(self[i]) = ya[i] for all i). Also return an error estimate on the returned value.

polisirreducible()
f.polisirreducible(): Returns True if f is an irreducible non-constant polynomial, or False if f is reducible or constant.

pollead(*v=-1*)
self.pollead(v): leading coefficient of polynomial or series self, or self itself if self is a scalar. Error otherwise. With respect to the main variable of self if v is omitted, with respect to the variable v otherwise

polrecip()

polred(*flag=0, fa=None*)

polredabs(*flag=0*)

polredbest(*flag=0*)

polresultant(*y, var=-1, flag=0*)

polroots(*flag=0, precision=0*)
polroots(x,flag=0): complex roots of the polynomial x. flag is optional, and can be 0: default, uses Schonhage's method modified by Gourdon, or 1: uses a modified Newton method.

polrootsmod(*p, flag=0*)

polrootspadic(*p, r=20*)

polrootspadicfast(*p, r=20*)

polsturm(*a, b*)

polsturm_full()

polsylvestermatrix(*g*)

polsym(*n*)

polylog(*x, m, flag=0, precision=0*)
x.polylog(m,flag=0): m-th polylogarithm of x. flag is optional, and can be 0: default, 1: D_m -modified m-th polylog of x, 2: D_m-modified m-th polylog of x, 3: P_m-modified m-th polylog of x.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

TODO: Add more explanation, copied from the PARI manual.

EXAMPLES:

```
sage: pari(10).polylog(3)
5.64181141475134 - 8.32820207698027*I
sage: pari(10).polylog(3,0)
5.64181141475134 - 8.32820207698027*I
sage: pari(10).polylog(3,1)
0.523778453502411
```

```
sage: pari(10).polylog(3,2)
-0.400459056163451
```

pr_get_e()

Returns the ramification index (over $\mathbf{Q}$) of this prime ideal.

NOTE: `self` must be a PARI prime ideal (as returned by `idealfactor` for example).

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: pari(K).idealfactor(K.ideal(2))[0,0].pr_get_e()
2
sage: pari(K).idealfactor(K.ideal(3))[0,0].pr_get_e()
1
sage: pari(K).idealfactor(K.ideal(5))[0,0].pr_get_e()
1
```

pr_get_f()

Returns the residue class degree (over $\mathbf{Q}$) of this prime ideal.

NOTE: `self` must be a PARI prime ideal (as returned by `idealfactor` for example).

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: pari(K).idealfactor(K.ideal(2))[0,0].pr_get_f()
1
sage: pari(K).idealfactor(K.ideal(3))[0,0].pr_get_f()
2
sage: pari(K).idealfactor(K.ideal(5))[0,0].pr_get_f()
1
```

pr_get_gen()

Returns the second generator of this PARI prime ideal, where the first generator is `self.pr_get_p()`.

NOTE: `self` must be a PARI prime ideal (as returned by `idealfactor` for example).

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: g = pari(K).idealfactor(K.ideal(2))[0,0].pr_get_gen(); g; K(g)
[1, 1]~
i + 1
sage: g = pari(K).idealfactor(K.ideal(3))[0,0].pr_get_gen(); g; K(g)
[3, 0]~
3
sage: g = pari(K).idealfactor(K.ideal(5))[0,0].pr_get_gen(); g; K(g)
[-2, 1]~
i - 2
```

pr_get_p()

Returns the prime of $\mathbf{Z}$ lying below this prime ideal.

NOTE: `self` must be a PARI prime ideal (as returned by `idealfactor` for example).

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: F = pari(K).idealfactor(K.ideal(5)); F
[[5, [-2, 1]~, 1, 1, [2, 1]~], 1; [5, [2, 1]~, 1, 1, [-2, 1]~], 1]
sage: F[0,0].pr_get_p()
5
```

precision(*x*, *n=-1*)

precision(x,n): Change the precision of x to be n, where n is a C-integer). If n is omitted, output the real precision of x.

INPUT:

- x - gen
- n - (optional) int

OUTPUT: nothing or gen if n is omitted

EXAMPLES:

primepi()

Return the number of primes less than or equal to self.

EXAMPLES:

```
sage: pari(7).primepi()
4
sage: pari(100).primepi()
25
sage: pari(1000).primepi()
168
sage: pari(100000).primepi()
9592
sage: pari(0).primepi()
0
sage: pari(-15).primepi()
0
sage: pari(500509).primepi()
41581
```

printtex(*x*)

psi(*x*, *precision=0*)

x.psi(): psi-function at x.

Return the ψ-function of x, i.e., the logarithmic derivative $\Gamma'(x)/\Gamma(x)$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).psi()
-0.577215664901533
```

python(*locals=None*)

Return Python eval of self.

Note: is self is a real (type t_REAL) the result will be a RealField element of the equivalent precision; if self is a complex (type t_COMPLEX) the result will be a ComplexField element of precision the minimum precision of the real and imaginary parts.

EXAMPLES:

```
sage: pari('389/17').python()
389/17
sage: f = pari('(2/3)*x^3 + x - 5/7 + y'); f
2/3*x^3 + x + (y - 5/7)
sage: var('x,y')
```

```
(x, y)
sage: f.python({'x':x, 'y':y})
2/3*x^3 + x + y - 5/7
```

You can also use .sage, which is a psynonym:

```
sage: f.sage({'x':x, 'y':y})
2/3*x^3 + x + y - 5/7
```

python_list()

Return a Python list of the PARI gens. This object must be of type t_VEC.

INPUT: None

OUTPUT:

- `list` - Python list whose elements are the elements of the input gen.

EXAMPLES:

```
sage: v=pari([1,2,3,10,102,10])
sage: w = v.python_list()
sage: w
[1, 2, 3, 10, 102, 10]
sage: type(w[0])
<type 'sage.libs.pari.gen.gen'>
sage: pari("[1,2,3]").python_list()
[1, 2, 3]
```

python_list_small()

Return a Python list of the PARI gens. This object must be of type t_VECSMALL, and the resulting list contains python 'int's.

EXAMPLES:

```
sage: v=pari([1,2,3,10,102,10]).Vecsmall()
sage: w = v.python_list_small()
sage: w
[1, 2, 3, 10, 102, 10]
sage: type(w[0])
<type 'int'>
```

qfbhclassno(*n*)

Computes the Hurwitz-Kronecker class number of n.

INPUT:

- n (gen) – a non-negative integer

Note: If n is large (more than $5 * 10^5$), the result is conditional upon GRH.

EXAMPLES:

The Hurwitz class number is 0 is n is congruent to 1 or 2 modulo 4:: sage: pari(-10007).qfbhclassno() 0 sage: pari(-2).qfbhclassno() 0

It is -1/12 for n=0:

```
sage: pari(0).qfbhclassno()
-1/12
```

Otherwise it is the number of classes of positive definite binary quadratic forms with discriminant $-n$, weighted by $1/m$ where m is the number of automorphisms of the form:

```
sage: pari(4).qfbhclassno()
1/2
sage: pari(3).qfbhclassno()
1/3
sage: pari(23).qfbhclassno()
3
```

qflll (*flag=0*)

qflll(x,flag=0): LLL reduction of the vectors forming the matrix x (gives the unimodular transformation matrix). The columns of x must be linearly independent, unless specified otherwise below. flag is optional, and can be 0: default, 1: assumes x is integral, columns may be dependent, 2: assumes x is integral, returns a partially reduced basis, 4: assumes x is integral, returns [K,I] where K is the integer kernel of x and I the LLL reduced image, 5: same as 4 but x may have polynomial coefficients, 8: same as 0 but x may have polynomial coefficients.

qflllgram (*flag=0*)

qflllgram(x,flag=0): LLL reduction of the lattice whose gram matrix is x (gives the unimodular transformation matrix). flag is optional and can be 0: default,1: lllgramint algorithm for integer matrices, 4: lllgramkerim giving the kernel and the LLL reduced image, 5: lllgramkerimgen same when the matrix has polynomial coefficients, 8: lllgramgen, same as qflllgram when the coefficients are polynomials.

qfminim (*b=None*, *m=None*, *flag=0*, *precision=0*)

Return vectors with bounded norm for this quadratic form.

INPUT:

- `self` – a quadratic form
- `b` – a bound on vector norm (finds minimal non-zero vectors if b=0)
- `m` – maximum number of vectors to return. If `None` (default), return all vectors of norm at most B
- flag (optional) –
  - 0: default;
  - 1: return only the first minimal vector found (ignore `max`);
  - 2: as 0 but uses a more robust, slower implementation, valid for non integral quadratic forms.

OUTPUT:

A triple consisting of

- the number of vectors of norm <= b,
- the actual maximum norm of vectors listed
- a matrix whose columns are vectors with norm less than or equal to b for the definite quadratic form. Only one of v and $-v$ is returned and the zero vector is never returned.

Note: If max is specified then only max vectors will be output, but all vectors withing the given norm bound will be computed.

EXAMPLES:

```
sage: A = Matrix(3,3,[1,2,3,2,5,5,3,5,11])
sage: A.is_positive_definite()
True
```

The first 5 vectors of norm at most 10:

```
sage: pari(A).qfminim(10, 5).python()
[
          [-17 -14 -15 -16 -13]
          [  4   3   3   3   2]
146, 10, [  3   3   3   3   3]
]
```

All vectors of minimal norm:

```
sage: pari(A).qfminim(0).python()
[
       [-5 -2  1]
       [ 1  1  0]
6, 1, [ 1  0  0]
]
```

Use flag=2 for non-integral input:

```
sage: pari(A.change_ring(RR)).qfminim(5, m=5, flag=2).python()
[
                            [ -5 -10  -2  -7   3]
                            [  1   2   1   2   0]
10, 5.00000000023283..., [  1   2   0   1  -1]
]
```

qfrep(*B*, *flag=0*)

qfrep(x,B,flag=0): vector of (half) the number of vectors of norms from 1 to B for the integral and definite quadratic form x. Binary digits of flag mean 1: count vectors of even norm from 1 to 2B, 2: return a t_VECSMALL instead of a t_VEC.

quadhilbert()

Returns a polynomial over $\mathbf{Q}$ whose roots generate the Hilbert class field of the quadratic field of discriminant `self` (which must be fundamental).

EXAMPLES:

```
sage: pari(-23).quadhilbert()
x^3 - x^2 + 1
sage: pari(145).quadhilbert()
x^4 - x^3 - 3*x^2 + x + 1
sage: pari(-12).quadhilbert()    # Not fundamental
Traceback (most recent call last):
...
PariError: quadray needs a fundamental discriminant
```

random(*N*)

`random(N=2^31)`: Return a pseudo-random integer between 0 and $N - 1$.

INPUT:

-`N` - gen, integer

OUTPUT:

- `gen` - integer

EXAMPLES:

real(*x*)

real(x): Return the real part of x.

INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

reverse()
Return the polynomial obtained by reversing the coefficients of this polynomial.

rnfcharpoly(*T*, *a*, *v='x'*)

rnfdisc(*x*)

rnfeltabstorel(*x*)

rnfeltreltoabs(*x*)

rnfequation(*poly*, *flag=0*)

rnfidealabstorel(*x*)

rnfidealdown(*x*)
rnfidealdown(rnf,x): finds the intersection of the ideal x with the base field.

EXAMPLES: sage: x = ZZ['xx1'].0; pari(x) xx1 sage: y = ZZ['yy1'].0; pari(y) yy1 sage: nf = pari(y^2 - 6*y + 24).nfinit() sage: rnf = nf.rnfinit(x^2 - pari(y))

This is the relative HNF of the inert ideal (2) in rnf:

```
sage: P = pari('[[[1, 0]~, [0, 0]~; [0, 0]~, [1, 0]~], [[2, 0; 0, 2], [2, 0; 0, 1/2]]]')
```

And this is the HNF of the inert ideal (2) in nf:

sage: rnf.rnfidealdown(P) [2, 0; 0, 2]

rnfidealhnf(*x*)

rnfidealnormrel(*x*)

rnfidealreltoabs(*x*)

rnfidealtwoelt(*x*)

rnfinit(*poly*)
EXAMPLES: We construct a relative number field.

```
sage: f = pari('y^3+y+1')
sage: K = f.nfinit()
sage: x = pari('x'); y = pari('y')
sage: g = x^5 - x^2 + y
sage: L = K.rnfinit(g)
```

rnfisfree(*poly*)

rnfisnorm(*T*, *flag=0*)

rnfisnorminit(*polrel*, *flag=2*)

round(*x*, *estimate=False*)
round(x,estimate=False): If x is a real number, returns x rounded to the nearest integer (rounding up). If the optional argument estimate is True, also returns the binary exponent e of the difference between the original and the rounded value (the "fractional part") (this is the integer ceiling of log_2(error)).

When x is a general PARI object, this function returns the result of rounding every coefficient at every level of PARI object. Note that this is different than what the truncate function does (see the example below).

One use of round is to get exact results after a long approximate computation, when theory tells you that the coefficients must be integers.

INPUT:

- `x` - gen
- `estimate` - (optional) bool, False by default

OUTPUT:

- if estimate is False, return a single gen.
- if estimate is True, return rounded version of x and error estimate in bits, both as gens.

EXAMPLES:

```
sage: pari('1.5').round()
2
sage: pari('1.5').round(True)
(2, -1)
sage: pari('1.5 + 2.1*I').round()
2 + 2*I
sage: pari('1.0001').round(True)
(1, -14)
sage: pari('(2.4*x^2 - 1.7)/x').round()
(2*x^2 - 2)/x
sage: pari('(2.4*x^2 - 1.7)/x').truncate()
2.40000000000000*x
```

sage (*locals=None*)

Return Python eval of self.

Note: is self is a real (type t_REAL) the result will be a RealField element of the equivalent precision; if self is a complex (type t_COMPLEX) the result will be a ComplexField element of precision the minimum precision of the real and imaginary parts.

EXAMPLES:

```
sage: pari('389/17').python()
389/17
sage: f = pari('(2/3)*x^3 + x - 5/7 + y'); f
2/3*x^3 + x + (y - 5/7)
sage: var('x,y')
(x, y)
sage: f.python({'x':x, 'y':y})
2/3*x^3 + x + y - 5/7
```

You can also use .sage, which is a psynonym:

```
sage: f.sage({'x':x, 'y':y})
2/3*x^3 + x + y - 5/7
```

serconvol (*g*)

serlaplace ()

serreverse ()

serreverse(f): reversion of the power series f.

If f(t) is a series in t with valuation 1, find the series g(t) such that g(f(t)) = t.

EXAMPLES:

```
sage: f = pari('x+x^2+x^3+O(x^4)'); f
x + x^2 + x^3 + O(x^4)
sage: g = f.serreverse(); g
x - x^2 + x^3 + O(x^4)
sage: f.subst('x',g)
x + O(x^4)
sage: g.subst('x',f)
x + O(x^4)
```

shift (*x*, *n*)
shift(x,n): shift x left n bits if n=0, right -n bits if n0.

shiftmul (*x*, *n*)
shiftmul(x,n): Return the product of x by 2^n.

sign (*x*)
Return the sign of x, where x is of type integer, real or fraction.

EXAMPLES:

```
sage: pari(pi).sign()
1
sage: pari(0).sign()
0
sage: pari(-1/2).sign()
-1
```

PARI throws an error if you attempt to take the sign of a complex number:

```
sage: pari(I).sign()
Traceback (most recent call last):
...
PariError: incorrect type in gsigne
```

simplify (*x*)
simplify(x): Simplify the object x as much as possible, and return the result.

A complex or quadratic number whose imaginary part is an exact 0 (i.e., not an approximate one such as O(3) or 0.E-28) is converted to its real part, and a a polynomial of degree 0 is converted to its constant term. Simplification occurs recursively.

This function is useful before using arithmetic functions, which expect integer arguments:

EXAMPLES:

```
sage: y = pari('y')
sage: x = pari('9') + y - y
sage: x
9
sage: x.type()
't_POL'
sage: x.factor()
matrix(0,2)
sage: pari('9').factor()
Mat([3, 2])
sage: x.simplify()
9
sage: x.simplify().factor()
Mat([3, 2])
sage: x = pari('1.5 + 0*I')
sage: x.type()
```

```
't_REAL'
sage: x.simplify()
1.50000000000000
sage: y = x.simplify()
sage: y.type()
't_REAL'
```

sin(*x*, *precision=0*)

x.sin(): The sine of x.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).sin()
0.841470984807897
sage: C.<i> = ComplexField()
sage: pari(1+i).sin()
1.29845758141598 + 0.634963914784736*I
```

sinh(*x*, *precision=0*)

The hyperbolic sine function.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(0).sinh()
0.E-19
sage: C.<i> = ComplexField()
sage: pari(1+i).sinh()
0.634963914784736 + 1.29845758141598*I
```

sizebyte(*x*)

Return the total number of bytes occupied by the complete tree of the object x. Note that this number depends on whether the computer is 32-bit or 64-bit.

INPUT:

- x - gen

OUTPUT: int (a Python int)

EXAMPLE:

```
sage: pari('1').sizebyte()
12            # 32-bit
24            # 64-bit
```

sizedigit(*x*)

sizedigit(x): Return a quick estimate for the maximal number of decimal digits before the decimal point of any component of x.

INPUT:

- x - gen

OUTPUT:

- `int` - Python integer

EXAMPLES:

```
sage: x = pari('10^100')
sage: x.Str().length()
101
sage: x.sizedigit()
101
```

Note that digits after the decimal point are ignored.

```
sage: x = pari('1.234')
sage: x
1.23400000000000
sage: x.sizedigit()
1
```

The estimate can be one too big:

```
sage: pari('7234.1').sizedigit()
4
sage: pari('9234.1').sizedigit()
5
```

sizeword(*x*)

Return the total number of machine words occupied by the complete tree of the object x. A machine word is 32 or 64 bits, depending on the computer.

INPUT:

- x - gen

OUTPUT: int (a Python int)

EXAMPLES:

```
sage: pari('0').sizeword()
2
sage: pari('1').sizeword()
3
sage: pari('1000000').sizeword()
3
sage: pari('10^100').sizeword()
13      # 32-bit
8       # 64-bit
sage: pari(RDF(1.0)).sizeword()
4       # 32-bit
3       # 64-bit
sage: pari('x').sizeword()
9
sage: pari('x^20').sizeword()
66
sage: pari('[x, I]').sizeword()
20
```

sqr(*x*)

x.sqr(): square of x. Faster than, and most of the time (but not always - see the examples) identical to x*x.

EXAMPLES:

```
sage: pari(2).sqr()
4
```

For 2-adic numbers, x.sqr() may not be identical to x*x (squaring a 2-adic number increases its precision):

```
sage: pari("1+O(2^5)").sqr()
1 + O(2^6)
sage: pari("1+O(2^5)")*pari("1+O(2^5)")
1 + O(2^5)
```

However:

```
sage: x = pari("1+O(2^5)"); x*x
1 + O(2^6)
```

sqrt (*x*, *precision=0*)

x.sqrt(precision): The square root of x.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).sqrt()
1.41421356237310
```

sqrtint (*x*)

Return the integer square root of the integer x, rounded towards zero.

EXAMPLES:

```
sage: pari(8).sqrtint()
2
sage: pari(10^100).sqrtint()
100000000000000000000000000000000000000000000000000
```

sqrtn (*x*, *n*, *precision=0*)

x.sqrtn(n): return the principal branch of the n-th root of x, i.e., the one such that $\arg(\sqrt(x)) \in]-\pi/n, \pi/n]$. Also returns a second argument which is a suitable root of unity allowing one to recover all the other roots. If it was not possible to find such a number, then this second return value is 0. If the argument is present and no square root exists, return 0 instead of raising an error.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

Note: intmods (modulo a prime) and p-adic numbers are allowed as arguments.

INPUT:

- x - gen
- n - integer

OUTPUT:

- gen - principal n-th root of x
- gen - root of unity z that gives the other roots

EXAMPLES:

```
sage: s, z = pari(2).sqrtn(5)
sage: z
0.309016994374947 + 0.951056516295154*I
```

```
sage: s
1.14869835499704
sage: s^5
2.00000000000000
sage: z^5
1.00000000000000 + 5.42101086 E-19*I         # 32-bit
1.00000000000000 + 5.96311194867027 E-19*I   # 64-bit
sage: (s*z)^5
2.00000000000000 + 1.409462824 E-18*I        # 32-bit
2.00000000000000 + 9.21571846612679 E-19*I   # 64-bit
```

subst (*var*, *z*)
In `self`, replace the variable `var` by the expression z.

EXAMPLES:

```
sage: x = pari("x"); y = pari("y")
sage: f = pari('x^3 + 17*x + 3')
sage: f.subst(x, y)
y^3 + 17*y + 3
sage: f.subst(x, "z")
z^3 + 17*z + 3
sage: f.subst(x, "z")^2
z^6 + 34*z^4 + 6*z^3 + 289*z^2 + 102*z + 9
sage: f.subst(x, "x+1")
x^3 + 3*x^2 + 20*x + 21
sage: f.subst(x, "xyz")
xyz^3 + 17*xyz + 3
sage: f.subst(x, "xyz")^2
xyz^6 + 34*xyz^4 + 6*xyz^3 + 289*xyz^2 + 102*xyz + 9
```

substpol (*y*, *z*)

sumdiv (*n*)
Return the sum of the divisors of n.

EXAMPLES:

```
sage: pari(10).sumdiv()
18
```

sumdivk (*n*, *k*)
Return the sum of the k-th powers of the divisors of n.

EXAMPLES:

```
sage: pari(10).sumdivk(2)
130
```

tan (*x*, *precision=0*)
x.tan() - tangent of x

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).tan()
-2.18503986326152
sage: C.<i> = ComplexField()
```

```
sage: pari(i).tan()
0.761594155955765*I
```

tanh (*x*, *precision=0*)

x.tanh() - hyperbolic tangent of x

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).tanh()
0.761594155955765
sage: C.<i> = ComplexField()
sage: z = pari(i); z
1.00000000000000*I
sage: result = z.tanh()
sage: result.real() <= 1e-18
True
sage: result.imag()
1.55740772465490
```

taylor (*v=-1*)

teichmuller (*x*)

teichmuller(x): teichmuller character of p-adic number x.

This is the unique $(p-1)$-st root of unity congruent to $x/p^{v_p(x)}$ modulo p.

EXAMPLES:

```
sage: pari('2+O(7^5)').teichmuller()
2 + 4*7 + 6*7^2 + 3*7^3 + O(7^5)
```

theta (*q*, *z*, *precision=0*)

q.theta(z): Jacobi sine theta-function.

If q or z is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(0.5).theta(2)
1.63202590295260
```

thetanullk (*q*, *k*, *precision=0*)

q.thetanullk(k): return the k-th derivative at z=0 of theta(q,z).

If q is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If q is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(0.5).thetanullk(1)
0.548978532560341
```

thue (*rhs*, *ne*)

thueinit (*flag=0*, *precision=0*)

trace()

Return the trace of this PARI object.

EXAMPLES:

```
sage: pari('[1,2; 3,4]').trace()
5
```

truncate(*x*, *estimate=False*)

truncate(x,estimate=False): Return the truncation of x. If estimate is True, also return the number of error bits.

When x is in the real numbers, this means that the part after the decimal point is chopped away, e is the binary exponent of the difference between the original and truncated value (the "fractional part"). If x is a rational function, the result is the integer part (Euclidean quotient of numerator by denominator) and if requested the error estimate is 0.

When truncate is applied to a power series (in X), it transforms it into a polynomial or a rational function with denominator a power of X, by chopping away the $O(X^k)$. Similarly, when applied to a p-adic number, it transforms it into an integer or a rational number by chopping away the $O(p^k)$.

INPUT:

- `x` - gen
- `estimate` - (optional) bool, which is False by default

OUTPUT:

- if estimate is False, return a single gen.
- if estimate is True, return rounded version of x and error estimate in bits, both as gens.

EXAMPLES:

```
sage: pari('(x^2+1)/x').round()
(x^2 + 1)/x
sage: pari('(x^2+1)/x').truncate()
x
sage: pari('1.043').truncate()
1
sage: pari('1.043').truncate(True)
(1, -5)
sage: pari('1.6').truncate()
1
sage: pari('1.6').round()
2
sage: pari('1/3 + 2 + 3^2 + O(3^3)').truncate()
34/3
sage: pari('sin(x+O(x^10))').truncate()
1/362880*x^9 - 1/5040*x^7 + 1/120*x^5 - 1/6*x^3 + x
sage: pari('sin(x+O(x^10))').round()   # each coefficient has abs < 1
x + O(x^10)
```

type()

Return the PARI type of self as a string.

Note: In Cython, it is much faster to simply use typ(self.g) for checking PARI types.

EXAMPLES:

```
sage: pari(7).type()
't_INT'
sage: pari('x').type()
't_POL'
```

valuation(*x*, *p*)

valuation(x,p): Return the valuation of x with respect to p.

The valuation is the highest exponent of p dividing x.

- If p is an integer, x must be an integer, an intmod whose modulus is divisible by p, a rational number, a p-adic number, or a polynomial or power series in which case the valuation is the minimum of the valuations of the coefficients.
- If p is a polynomial, x must be a polynomial or a rational function. If p is a monomial then x may also be a power series.
- If x is a vector, complex or quadratic number, then the valuation is the minimum of the component valuations.
- If x = 0, the result is $2^31 - 1$ on 32-bit machines or $2^63 - 1$ on 64-bit machines if x is an exact object. If x is a p-adic number or power series, the result is the exponent of the zero.

INPUT:

- `x` - gen
- `p` - coercible to gen

OUTPUT:

- `gen` - integer

EXAMPLES:

```
sage: pari(9).valuation(3)
2
sage: pari(9).valuation(9)
1
sage: x = pari(9).Mod(27); x.valuation(3)
2
sage: pari('5/3').valuation(3)
-1
sage: pari('9 + 3*x + 15*x^2').valuation(3)
1
sage: pari([9,3,15]).valuation(3)
1
sage: pari('9 + 3*x + 15*x^2 + O(x^5)').valuation(3)
1

sage: pari('x^2*(x+1)^3').valuation(pari('x+1'))
3
sage: pari('x + O(x^5)').valuation('x')
1
sage: pari('2*x^2 + O(x^5)').valuation('x')
2

sage: pari(0).valuation(3)
2147483647            # 32-bit
9223372036854775807   # 64-bit
```

variable (*x*)

variable(x): Return the main variable of the object x, or p if x is a p-adic number.

This function raises a TypeError exception on scalars, i.e., on objects with no variable associated to them.

INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari('x^2 + x -2').variable()
x
sage: pari('1+2^3 + O(2^5)').variable()
2
sage: pari('x+y0').variable()
x
sage: pari('y0+z0').variable()
y0
```

vecextract (*y*, *z=None*)

self.vecextract(y,z): extraction of the components of the matrix or vector x according to y and z. If z is omitted, y designates columns, otherwise y corresponds to rows and z to columns. y and z can be vectors (of indices), strings (indicating ranges as in"1..10") or masks (integers whose binary representation indicates the indices to extract, from left to right 1, 2, 4, 8, etc.)

Note: This function uses the PARI row and column indexing, so the first row or column is indexed by 1 instead of 0.

vecmax (*x*)

vecmax(x): Return the maximum of the elements of the vector/matrix x.

vecmin (*x*)

vecmin(x): Return the maximum of the elements of the vector/matrix x.

weber (*x*, *flag=0*, *precision=0*)

x.weber(flag=0): One of Weber's f functions of x. flag is optional, and can be 0: default, function f(x)=exp(-i*Pi/24)*eta((x+1)/2)/eta(x) such that $j = (f^{24} - 16)^3/f^{24}$, 1: function f1(x)=eta(x/2)/eta(x) such that $j = (f1^24 + 16)^3/f2^{24}$, 2: function f2(x)=sqrt(2)*eta(2*x)/eta(x) such that $j = (f2^{24} + 16)^3/f2^{24}$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

TODO: Add further explanation from PARI manual.

EXAMPLES:

```
sage: C.<i> = ComplexField()
sage: pari(i).weber()
1.18920711500272 + 0.E-19*I                    # 32-bit
1.18920711500272 + 2.71050543121376 E-20*I   # 64-bit
sage: pari(i).weber(1)
1.09050773266526
sage: pari(i).weber(2)
1.09050773266526
```

xgcd (*x*, *y*)

Returns u,v,d such that d=gcd(x,y) and u*x+v*y=d.

EXAMPLES:

```
sage: pari(10).xgcd(15)
(5, -1, 1)
```

zeta (*s*, *precision=0*)

zeta(s): zeta function at s with s a complex or a p-adic number.

If s is a complex number, this is the Riemann zeta function $\zeta(s) = \sum_{n\geq 1} n^{-s}$, computed either using the Euler-Maclaurin summation formula (if s is not an integer), or using Bernoulli numbers (if s is a negative integer or an even nonnegative integer), or using modular forms (if s is an odd nonnegative integer).

If s is a p-adic number, this is the Kubota-Leopoldt zeta function, i.e. the unique continuous p-adic function on the p-adic integers that interpolates the values of $(1 - p^{-k})\zeta(k)$ at negative integers k such that $k \equiv 1 \pmod{p-1}$ if p is odd, and at odd k if $p = 2$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

INPUT:

- `s` - gen (real, complex, or p-adic number)

OUTPUT:

- `gen` - value of zeta at s.

EXAMPLES:

```
sage: pari(2).zeta()
1.64493406684823
sage: x = RR(pi)^2/6
sage: pari(x)
1.64493406684823
sage: pari(3).zeta()
1.20205690315959
sage: pari('1+5*7+2*7^2+O(7^3)').zeta()
4*7^-2 + 5*7^-1 + O(7^0)
```

znprimroot ()

Return a primitive root modulo self, whenever it exists.

This is a generator of the group $(\mathbf{Z}/n\mathbf{Z})^*$, whenever this group is cyclic, i.e. if $n = 4$ or $n = p^k$ or $n = 2p^k$, where p is an odd prime and k is a natural number.

INPUT:

- `self` - positive integer equal to 4, or a power of an odd prime, or twice a power of an odd prime

OUTPUT: gen

EXAMPLES:

```
sage: pari(4).znprimroot()
Mod(3, 4)
sage: pari(10007^3).znprimroot()
Mod(5, 1002101470343)
sage: pari(2*109^10).znprimroot()
Mod(236736367459211723407, 473472734918423446802)
```

`sage.libs.pari.gen.`**`init_pari_stack`**`(`*s=8000000*`)`

Deprecated, use `pari.allocatemem()` instead.

EXAMPLES:

```
sage: from sage.libs.pari.gen import init_pari_stack
sage: init_pari_stack()
doctest:...: DeprecationWarning: init_pari_stack() is deprecated; use pari.allocatemem() instead
See http://trac.sagemath.org/10018 for details.
sage: pari.stacksize()
8000000
```

CHAPTER

FOURTEEN

CYTHON WRAPPER FOR THE PARMA POLYHEDRA LIBRARY (PPL)

Cython wrapper for the Parma Polyhedra Library (PPL)

The Parma Polyhedra Library (PPL) is a library for polyhedral computations over **Q**. This interface tries to reproduce the C++ API as faithfully as possible in Cython/Sage. For example, the following C++ excerpt:

```
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:

```
sage: from sage.libs.ppl import Variable, Constraint_System, C_Polyhedron
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:

```
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:

```
sage: poly_from_generators == poly_from_constraints
True
sage: poly_from_generators >= poly_from_constraints
```

```
True
sage: poly_from_generators <  poly_from_constraints
False
sage: poly_from_constraints.minimized_generators()
Generator_System {point(0/1, 0/1), point(0/1, 3/1), point(3/1, 0/1), point(3/1, 3/1)}
sage: poly_from_constraints.minimized_constraints()
Constraint_System {-x0+3>=0, -x1+3>=0, x0>=0, x1>=0}
```

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:

  ```
  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 neccesarily 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:

```
sage: from sage.libs.ppl import Variable, Generator_System, point, C_Polyhedron
sage: basis = range(0,5)
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 above computation (using PPL) finishes without noticeable delay (timeit measures it to be 90 microseconds on sage.math). Below we do the same computation with cddlib, which needs more than 3 seconds on the same hardware:

```
sage: basis = range(0,5)
sage: gs = [ tuple(coeff) for coeff in Permutations(basis) ]
```

```
sage: Polyhedron(vertices=gs, backend='cdd')  # long time (3s on sage.math, 2011)
A 4-dimensional polyhedron in QQ^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 cooefficients.
- `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

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:

```
sage: from sage.libs.ppl import Constraint, Constraint_System, Generator, Generator_System, Vari
sage: x = Variable(0)
sage: y = Variable(1)
sage: C_Polyhedron( 5*x-2*y >=  x+y-1 )
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point, 1 ray, 1 line
sage: cs = Constraint_System()
sage: cs.insert( x >= 0 )
sage: cs.insert( y >= 0 )
sage: C_Polyhedron(cs)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point, 2 rays
```

```
sage: C_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(-x-y) )
sage: gs.insert( ray(x) )
sage: C_Polyhedron(gs)
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point, 1 ray
```

The empty and universe polyhedra are constructed like this:

```
sage: C_Polyhedron(3, 'empty')
The empty polyhedron in QQ^3
sage: C_Polyhedron(3, 'empty').constraints()
Constraint_System {-1==0}
sage: C_Polyhedron(3, 'universe')
The space-filling polyhedron in QQ^3
sage: C_Polyhedron(3, 'universe').constraints()
Constraint_System {}
```

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:

```
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 `Linear_Expression`. Do not attempt to manually construct constraints.

EXAMPLES:

```
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:

```
sage: 0 == 1    # watch out!
False
sage: Linear_Expression(0) == 1
-1==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: ineq = (3*x+2*y+1>=0)
sage: ineq.OK()
True
```

ascii_dump()

Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd  = 'from sage.libs.ppl import Linear_Expression, Variable\n'
sage: sage_cmd += 'x = Variable(0)\n'
sage: sage_cmd += 'y = Variable(1)\n'
sage: sage_cmd += 'e = (3*x+2*y+1 > 0)\n'
sage: sage_cmd += 'e.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
sage: print err  # long time
size 4 1 3 2 -1 > (NNC)
```

coefficient(*v*)

Return the coefficient of the variable v.

INPUT:

- v – a Variable.

OUTPUT:

An integer.

EXAMPLES:

```
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:

```
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:

```
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:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: (x==0).is_equality()
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:

```
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:

```
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:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: (x==0).is_inequality()
False
sage: (x>=0).is_inequality()
True
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 an non-strict inequality constraint.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: (x==0).is_nonstrict_inequality()
```

```
False
sage: (x>=0).is_nonstrict_inequality()
True
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 an strict inequality constraint.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: (x==0).is_strict_inequality()
False
sage: (x>=0).is_strict_inequality()
False
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: $\sum 0x_i + 0 = 0$,

•a non-strict inequality: $\sum 0x_i + b \geq 0$ with $b \geq 0$, or

•a strict inequality: $\sum 0x_i + b > 0$ with $b > 0$.

OUTPUT:

Boolean. Returns `True` if and only if `self` is a tautological constraint.

EXAMPLES:

```
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:

```
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:

```
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:

```
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 )
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\n'
sage: sage_cmd += 'x = Variable(0)\n'
sage: sage_cmd += 'y = Variable(1)\n'
sage: sage_cmd += 'cs = Constraint_System( 3*x > 2*y+1 )\n'
sage: sage_cmd += 'cs.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
sage: print err  # long time
```

```
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:

```
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:

```
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:

```
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: Constraint_System_iterator(cs).next()
-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 $p = (\frac{a_0}{d}, \ldots, \frac{a_{n-1}}{d})^T$
- a closure point $c = (\frac{a_0}{d}, \ldots, \frac{a_{n-1}}{d})^T$

where n is the dimension of the space and, for points and closure points, d is the divisor.

INPUT/OUTPUT:

Use the helper functions line(), ray(), point(), and closure_point() to construct generators. Analogous class methods are also available, see Generator.line(), Generator.ray(), Generator.point(), Generator.closure_point(). Do not attempt to construct generators manually.

Note: The generators are constructed from linear expressions. The inhomogeneous term is always silently discarded.

EXAMPLES:

```
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)
```

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
```

ascii_dump()
Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd  = 'from sage.libs.ppl import Linear_Expression, Variable, point\n'
sage: sage_cmd += 'x = Variable(0)\n'
sage: sage_cmd += 'y = Variable(1)\n'
sage: sage_cmd += 'p = point(3*x+2*y)\n'
sage: sage_cmd += 'p.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
sage: print err  # long time
size 3 1 3 2 P (C)
```

static **closure_point** (*expression=0*, *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:

```
sage: from sage.libs.ppl import Generator, Variable
sage: y = Variable(1)
sage: Generator.closure_point(2*y+7, 3)
closure_point(0/3, 2/3)
sage: Generator.closure_point(y+7, 3)
closure_point(0/3, 1/3)
sage: Generator.closure_point(7, 3)
closure_point()
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:

```
sage: from sage.libs.ppl import Variable, line
sage: x = Variable(0)
sage: line = line(3*x+1)
sage: line
line(1)
```

```
sage: line.coefficient(x)
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:

```
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:

```
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:

```
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:

```
sage: 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
```

static **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.
```

static 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:

```
sage: from sage.libs.ppl import Generator, Variable
sage: y = Variable(1)
sage: Generator.point(2*y+7, 3)
point(0/3, 2/3)
sage: Generator.point(y+7, 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.
```

static 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:

```
sage: from sage.libs.ppl import Generator, Variable
sage: 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:

```
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:

```
sage: from sage.libs.ppl import Variable, point, closure_point, ray, line
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:

```
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:

```
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:

```
sage: sage_cmd  = 'from sage.libs.ppl import Generator_System, point, Variable\n'
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
sage: print err  # long time
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:

```
sage: from sage.libs.ppl import Variable, Generator_System, point
sage: x = Variable(0)
sage: gs = Generator_System( point(3*x) ); gs
Generator_System {point(3/1)}
sage: gs.clear()
sage: gs
Generator_System {}
```

empty()

Return `True` if and only if `self` has no generators.

OUTPUT:

Boolean.

EXAMPLES:

```
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:

```
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:

```
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:

```
sage: from sage.libs.ppl import Generator_System, Variable, line, ray, point, closure_point, Gen
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: Generator_System_iterator(gs).next()
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:

```
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:

```
sage: sage_cmd  = 'from sage.libs.ppl import Linear_Expression, Variable\n'
sage: sage_cmd += 'x = Variable(0)\n'
sage: sage_cmd += 'y = Variable(1)\n'
sage: sage_cmd += 'e = 3*x+2*y+1\n'
sage: sage_cmd += 'e.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
sage: print err  # long time
size 3 1 3 2
```

coefficient(*v*)

Return the coefficient of the variable `v`.

INPUT:

- `v` – a `Variable`.

OUTPUT:

An integer.

EXAMPLES:

```
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:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0);  y = Variable(1)
sage: e = 3*x+5*y+1
sage: e.coefficients()
(3, 5)
```

inhomogeneous_term()

Return the inhomogeneous term of the linear expression.

OUTPUT:

Integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Linear_Expression
sage: Linear_Expression(10).inhomogeneous_term()
10
```

is_zero()

Test if `self` is the zero linear expression.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, Linear_Expression
sage: Linear_Expression(0).is_zero()
True
sage: Linear_Expression(10).is_zero()
False
```

space_dimension()

Return the dimension of the vector space necessary for the linear expression.

OUTPUT:

Integer.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: y = Variable(1)
sage: ( x+y+1 ).space_dimension()
2
sage: ( x+y   ).space_dimension()
2
sage: (   y+1 ).space_dimension()
2
sage: ( x  +1 ).space_dimension()
1
sage: ( y+1-y ).space_dimension()
2
```

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:

```
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:

```
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:

```
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.optimal_value()
10/3
```

TESTS:

```
sage: z = Variable(2)
sage: m.add_constraint(z >= -3)
Traceback (most recent call last):
...
ValueError: PPL::MIP_Problem::add_constraint(c):
c.space_dimension() == 3 exceeds this->space_dimension == 2.
```

add_constraints(*cs*)

Adds a copy of the constraints in cs to the 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)
sage: m.set_objective_function(x + y)
sage: m.add_constraints(cs)
sage: m.optimal_value()
10/3
```

TESTS:

```
sage: p = Variable(9)
sage: cs.insert(p >= -3)
sage: m.add_constraints(cs)
Traceback (most recent call last):
...
ValueError: PPL::MIP_Problem::add_constraints(cs):
cs.space_dimension() == 10 exceeds this->space_dimension() == 2.
```

add_space_dimensions_and_embed(*m*)

Adds m new space dimensions and embeds the old MIP problem in the new vector space.

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.add_space_dimensions_and_embed(5)
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)
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:

```
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:

```
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:

```
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:

```
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.optimal_value()
10/3
```

TESTS:

```
sage: z = Variable(2)
sage: m.set_objective_function(x + y + z)
Traceback (most recent call last):
...
ValueError: PPL::MIP_Problem::set_objective_function(obj):
obj.space_dimension() == 3 exceeds this->space_dimension == 2.
```

set_optimization_mode(*mode*)

Sets the optimization mode to mode.

EXAMPLES:

```
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'
```

TESTS:

```
sage: m.set_optimization_mode('max')
Traceback (most recent call last):
...
ValueError: Unknown value: mode=max.
```

solve()

Optimizes the MIP_Problem

EXAMPLES:

```
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:

```
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.

OUTPUT:

A C_Polyhedron.

EXAMPLES:

```
sage: from sage.libs.ppl import Constraint, Constraint_System, Generator, Generator_System, Vari
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
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 ray
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:

```
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`**

Bases: `object`

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:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: saturates     = Poly_Con_Relation.saturates();  saturates
saturates
sage: is_included   = Poly_Con_Relation.is_included(); is_included
is_included
sage: is_included.implies(saturates)
False
sage: saturates.implies(is_included)
False
sage: rels = []
sage: rels.append( Poly_Con_Relation.nothing() )
sage: rels.append( Poly_Con_Relation.is_disjoint() )
sage: rels.append( Poly_Con_Relation.strictly_intersects() )
sage: rels.append( Poly_Con_Relation.is_included() )
sage: rels.append( Poly_Con_Relation.saturates() )
sage: rels
[nothing, is_disjoint, strictly_intersects, is_included, saturates]
sage: from sage.matrix.constructor import matrix
sage: m = matrix(5,5)
sage: for i, rel_i in enumerate(rels):
...       for j, rel_j in enumerate(rels):
...           m[i,j] = rel_i.implies(rel_j)
sage: m
[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]
```

OK(*check_non_empty=False*)

Check if all the invariants are satisfied.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.nothing().OK()
True
```

ascii_dump()

Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd  = 'from sage.libs.ppl import Poly_Con_Relation\n'
sage: sage_cmd += 'Poly_Con_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
sage: print err  # long time
NOTHING
```

implies(*y*)

Test whether `self` implies `y`.

INPUT:

- `y` – a `Poly_Con_Relation`.

OUTPUT:

Boolean. `True` if and only if `self` implies `y`.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: nothing = Poly_Con_Relation.nothing()
sage: nothing.implies( nothing )
True
```

static **is_disjoint**()

Return the assertion "The polyhedron and the set of points satisfying the constraint are disjoint".

OUTPUT:

A `Poly_Con_Relation`.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.is_disjoint()
is_disjoint
```

static **is_included**()

Return the assertion "The polyhedron is included in the set of points satisfying the constraint".

OUTPUT:

A `Poly_Con_Relation`.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.is_included()
is_included
```

static **nothing**()

Return the assertion that says nothing.

OUTPUT:

A Poly_Con_Relation.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.nothing()
nothing
```

static saturates()

Return the assertion "".

OUTPUT:

A Poly_Con_Relation.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.saturates()
saturates
```

static strictly_intersects()

Return the assertion "The polyhedron intersects the set of points satisfying the constraint, but it is not included in it".

OUTPUT:

A Poly_Con_Relation.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Con_Relation
sage: Poly_Con_Relation.strictly_intersects()
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:

```
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:

```
sage: from sage.libs.ppl import Poly_Gen_Relation
sage: Poly_Gen_Relation.nothing().OK()
True
```

ascii_dump()

Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd  = 'from sage.libs.ppl import Poly_Gen_Relation\n'
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
sage: print err  # long time
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:

```
sage: from sage.libs.ppl import Poly_Gen_Relation
sage: nothing = Poly_Gen_Relation.nothing()
sage: nothing.implies( nothing )
True
```

static **nothing**()

Return the assertion that says nothing.

OUTPUT:

A `Poly_Gen_Relation`.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Gen_Relation
sage: Poly_Gen_Relation.nothing()
nothing
```

static **subsumes**()

Return the assertion "Adding the generator would not change the polyhedron".

OUTPUT:

A `Poly_Gen_Relation`.

EXAMPLES:

```
sage: from sage.libs.ppl import Poly_Gen_Relation
sage: Poly_Gen_Relation.subsumes()
subsumes
```

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:

```
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 alse `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:

```
    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 alse `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:

```
   sage: from sage.libs.ppl import Variable, C_Polyhedron, point, closure_point, ray
   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::

   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::

   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:

```
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 alse `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.

Raises 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:

```
   sage: from sage.libs.ppl import Variable, C_Polyhedron, Generator_System, point, ray, clo
   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,
:class:`C_Polyhedron` does not allow :func:`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

$$\Big\{(x,y,z)^T \in \mathbf{R}^3 \Big| (x,y)^T \in P\Big\}$$

INPUT:

- `m` – integer.

OUTPUT:

This method assigns the embedded polyhedron to `self` and does not return anything.

Raises 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

$$\left\{(x,y,0)^T \in \mathbf{R}^3 \middle| (x,y)^T \in P\right\}$$

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 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_project(1)
sage: p.minimized_generators()
Generator_System {point(3/1, 0/1)}
sage: p.add_space_dimensions_and_project( p.max_space_dimension() )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::add_space_dimensions_and_project(m):
adding m new space dimensions exceeds the maximum allowed space dimension.
```

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: 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.affine_dimension()
1
```

ascii_dump()

Write an ASCII dump to stderr.

EXAMPLES:

```
sage: sage_cmd  = 'from sage.libs.ppl import C_Polyhedron, Variable\n'
sage: sage_cmd += 'x = Variable(0)\n'
sage: sage_cmd += 'y = Variable(1)\n'
sage: sage_cmd += 'p = C_Polyhedron(3*x+2*y==1)\n'
sage: sage_cmd += 'p.minimized_generators()\n'
sage: sage_cmd += 'p.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
sage: print err  # long time
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 `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:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, Linear_Expression
sage: x = Variable(0);  y = Variable(1)
sage: p = C_Polyhedron(y<=0)
sage: p.bounds_from_above(x+1)
False
sage: p.bounds_from_above(Linear_Expression(y))
True
sage: p = C_Polyhedron(x<=0)
sage: p.bounds_from_above(y+1)
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:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, Linear_Expression
sage: x = Variable(0);  y = Variable(1)
sage: p = C_Polyhedron(y>=0)
sage: p.bounds_from_below(x+1)
False
sage: p.bounds_from_below(Linear_Expression(y))
True
sage: p = C_Polyhedron(x<=0)
sage: p.bounds_from_below(y+1)
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 functions returns the Cartiesian 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

$$R = \left\{ (x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1})^T \in \mathbf{R}^{n+m} \middle| (x_0, \ldots, x_{n-1})^T \in P, \ (y_0, \ldots, y_{m-1})^T \in Q \right\}$$

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: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron, point
sage: x = Variable(0)
sage: p1 = C_Polyhedron( point(1*x) )
sage: p2 = C_Polyhedron( point(2*x) )
sage: p1.concatenate_assign(p2)
sage: p1.minimized_generators()
Generator_System {point(1/1, 2/1)}
```

The polyhedra must be topology-compatible and not exceed the maximum space dimension:

```
sage: p1.concatenate_assign( NNC_Polyhedron(1, 'universe') )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::concatenate_assign(y):
y is a NNC_Polyhedron.
sage: p1.concatenate_assign( C_Polyhedron(p1.max_space_dimension(), 'empty') )
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: from sage.libs.ppl import Variable, C_Polyhedron
sage: x = Variable(0)
sage: p = C_Polyhedron(1, 'universe')
sage: p.constrains(x)
False
sage: p = C_Polyhedron(x>=0)
sage: p.constrains(x)
True
sage: y = Variable(1)
sage: p.constrains(y)
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:

```
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()
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 and `y` are topology-incompatible or dimension-incompatible.

EXAMPLES:

```
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:

```
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:

```
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.
```

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 removed.

Note: The modified polyhedron is not neccessarily a lattice polyhedron; Some vertices will, in general, still be rational. Lattice points interior to the polyhedron may be lost in the process.

EXAMPLES:

```
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:

```
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) )
sage: p.generators()
Generator_System {point(-1/1, -1/1, 0/1), point(0/1, 0/1, 0/1), point(1/1, 1/1, 0/1)}
sage: p.minimized_generators()
Generator_System {point(-1/1, -1/1, 0/1), point(1/1, 1/1, 0/1)}
```

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 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: 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.
sage: p.intersection_assign( NNC_Polyhedron(x+y<1) )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::intersection_assign(y):
y is a NNC_Polyhedron.
```

`is_bounded``()`

Test whether `self` is bounded.

OUTPUT:

Boolean. Returns `True` if and only if `self` is a bounded polyhedron.

EXAMPLES:

```
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:

```
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.

Rayises 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);  y = Variable(1)
sage: C_Polyhedron(x<=0).is_disjoint_from( C_Polyhedron(x>=1) )
True
```

This is not allowed:

```
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:

```
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:

```
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:

Returns True if and only if self is a topologically closed subset of the ambient vector space.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron
sage: x = Variable(0);  y = Variable(1)
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:

```
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:

```
sage: from sage.libs.ppl import C_Polyhedron
sage: C_Polyhedron(1, 'empty').max_space_dimension()    # random output
1152921504606846974
sage: C_Polyhedron(1, 'empty').max_space_dimension()
357913940            # 32-bit
1152921504606846974  # 64-bit
```

maximize(*expr*)

Maximize 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 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, I
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 )
{'sup_d': 3, 'sup_n': 10, 'bounded': True, 'maximum': True, 'generator': point(10/3, 0/3)}
```

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 )
{'sup_d': 1, 'sup_n': 0, 'bounded': True, 'maximum': False, 'generator': closure_point(0/1)}
```

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:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, NNC_Polyhedron, Constraint_System, I
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 )
{'minimum': True, 'bounded': True, 'inf_d': 1, 'generator': point(0/1, 0/1), 'inf_n': 0}
```

Unbounded case:

```
sage: cs = Constraint_System()
sage: cs.insert( x>0 )
sage: p = NNC_Polyhedron(cs)
sage: p.minimize( +x )
{'minimum': False, 'bounded': True, 'inf_d': 1, 'generator': closure_point(0/1), 'inf_n': 0}
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 )
sage: p.constraints()
Constraint_System {x1>=0, x0>=0, x0+x1>=0}
sage: p.minimized_constraints()
Constraint_System {x1>=0, x0>=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) )
sage: p.generators()
Generator_System {point(-1/1, -1/1, 0/1), point(0/1, 0/1, 0/1), point(1/1, 1/1, 0/1)}
sage: p.minimized_generators()
Generator_System {point(-1/1, -1/1, 0/1), point(1/1, 1/1, 0/1)}
```

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 and y are topology-incompatible or dimension-incompatible.

EXAMPLES:

```
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:

```
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:

```
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.
```

poly_hull_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 and 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)}
```

`self` and `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) )
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.
```

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(1*x+0*y) )
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):
...
ValueError: PPL::C_Polyhedron::relation_with(g):
this->space_dimension() == 2, g.space_dimension() == 3.
sage: p.relation_with( z>0 )
Traceback (most recent call last):
...
ValueError: PPL::C_Polyhedron::relation_with(c):
this->space_dimension() == 2, c.space_dimension() == 3.
```

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:

```
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:

```
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:

```
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:

```
sage: from sage.libs.ppl import Variable, C_Polyhedron, point
sage: x = Variable(0)
sage: y = Variable(1)
sage: p = C_Polyhedron( point(x+y) ); p
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point
sage: p.unconstrain(x); p
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point, 1 line
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.
```

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 and `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)}
```

`self` and `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) )
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.
```

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
sage: x = Variable(123)
sage: x.id()
123
sage: 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 `e1` and `e2` are equivalent, since the two variables `x` and `z` denote the same Cartesian axis:

```
sage: x = Variable(0)
sage: y = Variable(1)
sage: z = Variable(0)
sage: e1 = x + y; e1
x0+x1
sage: e2 = y + z; e2
x0+x1
sage: e1 - e2
0
```

OK()

Checks if all the invariants are satisfied.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(0)
sage: x.OK()
True
```

id()

Return the index of the Cartesian axis associated to the variable.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable
sage: x = Variable(123)
sage: x.id()
123
```

space_dimension()

Return the dimension of the vector space enclosing `self`.

OUPUT:

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
```

`sage.libs.ppl.`**closure_point**(*expression=0*, *divisor=1*)

Constuct 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.libs.ppl.`**equation**(*expression*)

Constuct an equation.

INPUT:

- `expression` – a `Linear_Expression`.

OUTPUT:

The equation `expression` == 0.

EXAMPLES:

```
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*)

Constuct an inequality.

INPUT:

- `expression` – a `Linear_Expression`.

OUTPUT:

The inequality `expression` >= 0.

EXAMPLES:

```
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*)

Constuct a line.

See `Generator.line()` for documentation.

EXAMPLES:

```
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*)

Constuct a point.

See `Generator.point()` for documentation.

EXAMPLES:

```
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*)

Constuct a ray.

See `Generator.ray()` for documentation.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, ray
sage: y = Variable(1)
sage: ray(2*y)
ray(0, 1)
```

`sage.libs.ppl.`**`strict_inequality`**(*expression*)

Constuct a strict inequality.

INPUT:

- `expression` – a `Linear_Expression`.

OUTPUT:

The inequality `expression` > 0.

EXAMPLES:

```
sage: from sage.libs.ppl import Variable, strict_inequality
sage: y = Variable(1)
sage: 2*y+1 > 0
2*x1+1>0
sage: strict_inequality(2*y+1)
2*x1+1>0
```

CHAPTER

FIFTEEN

INTERFACE TO THE PSELECT() SYSTEM CALL

Interface to the `pselect()` system call

This module defines a class `PSelecter` which can be used to call the system call `pselect()` and which can also be used in a `with` statement to block given signals until `PSelecter.pselect()` is called.

AUTHORS:

- Jeroen Demeyer (2013-02-07): initial version (trac ticket #14079)

15.1 Waiting for subprocesses

One possible use is to wait with a **timeout** until **any child process** exits, as opposed to `os.wait()` which doesn't have a timeout or `multiprocessing.Process.join()` which waits for one specific process.

Since `SIGCHLD` is ignored by default, we first need to install a signal handler for `SIGCHLD`. It doesn't matter what it does, as long as the signal isn't ignored:

```
sage: import signal
sage: def dummy_handler(sig, frame):
....:     pass
....:
sage: _ = signal.signal(signal.SIGCHLD, dummy_handler)
```

We wait for a child created using the `subprocess` module:

```
sage: from sage.ext.pselect import PSelecter
sage: from subprocess import *
sage: with PSelecter([signal.SIGCHLD]) as sel:
....:     p = Popen(["sleep", "1"])
....:     _ = sel.sleep()
....:
sage: p.poll()  # p should be finished
0
```

Now using the `multiprocessing` module:

```
sage: from sage.ext.pselect import PSelecter
sage: from multiprocessing import *
sage: import time
sage: with PSelecter([signal.SIGCHLD]) as sel:
```

```
....:     p = Process(target=time.sleep, args=(1,))
....:     p.start()
....:     _ = sel.sleep()
....:     p.is_alive()  # p should be finished
....:
False
```

class sage.ext.pselect.**PSelecter**

Bases: object

This class gives an interface to the pselect system call.

It can be used in a with statement to block given signals such that they can only occur during the pselect() or sleep() calls.

As an example, we block the SIGHUP and SIGALRM signals and then raise a SIGALRM signal. The interrupt will only be seen during the sleep() call:

```
sage: from sage.ext.pselect import PSelecter
sage: import signal, time
sage: with PSelecter([signal.SIGHUP, signal.SIGALRM]) as sel:
....:     os.kill(os.getpid(), signal.SIGALRM)
....:     time.sleep(0.5)  # Simply sleep, no interrupt detected
....:     try:
....:         _ = sel.sleep(1)  # Interrupt seen here
....:     except AlarmInterrupt:
....:         print("Interrupt OK")
....:
Interrupt OK
```

Warning: If SIGCHLD is blocked inside the with block, then you should not use Popen().wait() or Process().join() because those might block, even if the process has actually exited. Use non-blocking alternatives such as Popen.poll() or multiprocessing.active_children() instead.

pselect(*rlist*=[ ], *wlist*=[ ], *xlist*=[ ], *timeout*=*None*)

Wait until one of the given files is ready, or a signal has been received, or until timeout seconds have past.

INPUT:

- rlist – (default: []) a list of files to wait for reading.
- wlist – (default: []) a list of files to wait for writing.
- xlist – (default: []) a list of files to wait for exceptions.
- timeout – (default: None) a timeout in seconds, where None stands for no timeout.

OUTPUT: A 4-tuple (rready, wready, xready, tmout) where the first three are lists of file descriptors which are ready, that is a subset of (rlist, wlist, xlist). The fourth is a boolean which is True if and only if the command timed out. If pselect was interrupted by a signal, the output is ([], [], [], False).

See Also:

Use the sleep() method instead if you don't care about file descriptors.

EXAMPLES:

The file /dev/null should always be available for reading and writing:

```
sage: from sage.ext.pselect import PSelecter
sage: f = open(os.devnull, "r+")
sage: sel = PSelecter()
sage: sel.pselect(rlist=[f])
([<open file '/dev/null', mode 'r+' at ...>], [], [], False)
sage: sel.pselect(wlist=[f])
([], [<open file '/dev/null', mode 'r+' at ...>], [], False)
```

A list of various files, all of them should be ready for reading. Also create a pipe, which should be ready for writing, but not reading (since nothing has been written):

```
sage: f = open(os.devnull, "r")
sage: g = open(os.path.join(SAGE_LOCAL, 'bin', 'python'), "r")
sage: (pr, pw) = os.pipe()
sage: r, w, x, t = PSelecter().pselect([f,g,pr,pw], [pw], [pr,pw])
sage: len(r), len(w), len(x), t
(2, 1, 0, False)
```

Checking for exceptions on the pipe should simply time out:

```
sage: sel.pselect(xlist=[pr,pw], timeout=0.2)
([], [], [], True)
```

TESTS:

It is legal (but silly) to list the same file multiple times:

```
sage: r, w, x, t = PSelecter().pselect([f,g,f,f,g])
sage: len(r)
5
```

Invalid input:

```
sage: PSelecter().pselect([None])
Traceback (most recent call last):
...
TypeError: an integer is required
```

Open a file and close it, but save the (invalid) file descriptor:

```
sage: f = open(os.devnull, "r")
sage: n = f.fileno()
sage: f.close()
sage: PSelecter().pselect([n])
Traceback (most recent call last):
...
IOError: ...
```

sleep (*timeout=None*)

Wait until a signal has been received, or until `timeout` seconds have past.

This is implemented as a special case of `pselect()` with empty lists of file descriptors.

INPUT:

- `timeout` – (default: `None`) a timeout in seconds, where `None` stands for no timeout.

OUTPUT: A boolean which is `True` if the call timed out, False if it was interrupted.

EXAMPLES:

A simple wait with timeout:

```
sage: from sage.ext.pselect import PSelecter
sage: sel = PSelecter()
sage: sel.sleep(timeout=0.1)
True
```

0 or negative time-outs are allowed, `sleep` should then return immediately:

```
sage: sel.sleep(timeout=0)
True
sage: sel.sleep(timeout=-123.45)
True
```

`sage.ext.pselect.`**`get_fileno`**(*f*)

Return the file descriptor of `f`.

INPUT:

- `f` – an object with a `.fileno` method or an integer, which is a file descriptor.

OUTPUT: A C `long` representing the file descriptor.

EXAMPLES:

```
sage: from sage.ext.pselect import get_fileno
sage: get_fileno(open(os.devnull))  # random
5
sage: get_fileno(42)
42
sage: get_fileno(None)
Traceback (most recent call last):
...
TypeError: an integer is required
sage: get_fileno(-1)
Traceback (most recent call last):
...
ValueError: Invalid file descriptor
sage: get_fileno(2^30)
Traceback (most recent call last):
...
ValueError: Invalid file descriptor
```

CHAPTER

SIXTEEN

LIBSINGULAR: FUNCTIONS

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:

```
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:

```
sage: from sage.libs.singular.function import singular_function, lib as singular_lib
sage: primdecSY = singular_function('primdecSY')
Traceback (most recent call last):
...
NameError: Function 'primdecSY' is not defined.
```

```
sage: singular_lib('primdec.lib')
sage: primdecSY = singular_function('primdecSY')
```

There is also a short-hand notation for the above:

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

The above line will load "primdec.lib" first and then load the function `primdecSY`.

TESTS:

```
sage: from sage.libs.singular.function import singular_function
sage: std = singular_function('std')
sage: loads(dumps(std)) == std
True
```

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.

class `sage.libs.singular.function.`**`Converter`**

Bases: `sage.structure.sage_object.SageObject`

A `Converter` interfaces between Sage objects and Singular interpreter objects.

`ring``()`

Return the ring in which the arguments of this list live.

EXAMPLE:

```
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
```

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.

This class implements calling a kernel function.

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

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.

This class implements calling a library function.

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

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.

EXAMPLE:

```
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()
0
```

`is_commutative``()`

Determine whether a given ring is commutative.

EXAMPLE:

```
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()
True
```

`ngens``()`

Get number of generators.

EXAMPLE:

```
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()
3
```

`npars``()`

Get number of parameters.

EXAMPLE:

```
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()
0
```

`ordering_string``()`

Get Singular string defining monomial ordering.

EXAMPLE:

```
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).ordering_string()
'dp(3),C'
```

par_names()

Get parameter names.

EXAMPLE:

```
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).par_names()
[]
```

var_names()

Get names of variables.

EXAMPLE:

```
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).var_names()
['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:

```
sage: from sage.libs.singular.function import SingularKernelFunction
sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: I = R.ideal(x, x+1)
sage: f = SingularKernelFunction("std")
sage: f(I)
[1]
```

class sage.libs.singular.function.**SingularLibraryFunction**

Bases: sage.libs.singular.function.SingularFunction

EXAMPLES:

```
sage: from sage.libs.singular.function import SingularLibraryFunction
sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: 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.

EXAMPLE:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
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`.

EXAMPLE:

```
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

EXAMPLE:

```
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: primes = singular_function('primes')
sage: primes(2,10, ring=GF(127)['x,y,z'])
(2, 3, 5, 7)
```

`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.

EXAMPLE:

```
sage: 'groebner' in sage.libs.singular.function.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:

```
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], ring=P)
3
sage: size("sage", ring=P)
4
sage: size(["hello", "sage"], ring=P)
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:

```
sage: factorize(ring=P)
Traceback (most recent call last):
...
```

```
RuntimeError: Error in Singular function call 'factorize':
 Wrong number of arguments
sage: factorize(f, 1, 2)
Traceback (most recent call last):
...
RuntimeError: Error in Singular function call 'factorize':
 Wrong number of arguments
sage: factorize(f, 1, 2, 3)
Traceback (most recent call last):
...
RuntimeError: Error in Singular function call 'factorize':
 Wrong number of arguments
```

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

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

We try to define a non-existing function:

```
sage: number_foobar = singular_function('number_foobar');
Traceback (most recent call last):
...
NameError: Function 'number_foobar' is not defined.

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,ring=P)
67957045707/25000000000
sage: RR(number_e(10r,ring=P))
2.71828182828000

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: l = ringlist(P)
sage: l[3].__class__
<class 'sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic'>
sage: l
[0, ['x', 'y', 'z'], [['dp', (1, 1, 1)], ['C', (0,)]], [0]]
sage: ring=singular_function("ring")
sage: ring(l, ring=P)
<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: F.<x,y,z> = GF(3)[]
sage: intmat = Matrix(ZZ, 2,2, [100,2,3,4])
sage: det(intmat, ring=F)
394
sage: random = singular_function("random")
sage: A = random(10,2,3, ring =F); 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: v=M((100*x,5*y,10*z*x*y))
sage: leadcoef(v)
10
sage: v = M([x+y,x*y+y**3,z])
sage: lead = singular_function("lead")
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-y, y*2,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)
[[[(-2*y, 2, y + 1, 0), (0, x + 1, 1, -y), (0, -2, x - 1, 0), (x*y - y, -y + 1, 1, -y), (x^2 + 1
sage: mres = singular_function("mres")
sage: resolution = mres(M, 0)
sage: resolution
<Resolution>
sage: singular_list(resolution)
[[(-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: 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")
sage: twostd(I)
[x + y, y^2]
sage: M=syz(I)
doctest...
sage: M
[(x + y, x*y)]
sage: syz(M, ring=P)
[(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, ring=P)
<noncommutative RingWrap>
sage: I=twostd(I)
sage: l[3]=I
sage: ring(l, ring=P)
<noncommutative RingWrap>
```

CHAPTER

SEVENTEEN

LIBSINGULAR: OPTIONS

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: std = sage.libs.singular.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:

```
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)

sage: print opt
general options for libSingular (current value 0x02000082)
```

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

```
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:

```
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 `notRegularity` - 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`.
- $red_s b$‘ 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 odering 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.

EXAMPLE:

```
sage: from sage.libs.singular.option import LibSingularOptions
sage: libsingular_options = LibSingularOptions()
sage: libsingular_options
general options for libSingular (current value 0x06000082)
```

Here we demonstrate the intended way of using libSingular options:

```
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]
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
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
```

reset_default()

Reset libSingular's default options.

EXAMPLE:

```
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.

EXAMPLE:

```
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
...
general options for libSingular (current value 0x04000082)
general options for libSingular (current value 0x04000002)

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*)

EXAMPLE:

```
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.

EXAMPLE:

```
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

Pythonic 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.

EXAMPLE:

```
sage: from sage.libs.singular.option import LibSingularVerboseOptions
sage: libsingular_verbose = LibSingularVerboseOptions()
sage: libsingular_verbose
verbosity options for libSingular (current value 0x00002851)
```

reset_default()

Return to libSingular's default verbosity options

EXAMPLE:

```
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()
sage: opt_verb['not_warn_sb']
False
```

CHAPTER

EIGHTEEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

BIBLIOGRAPHY

[PariUsers] User's Guide to PARI/GP, http://pari.math.u-bordeaux.fr/pub/pari/manuals/2.5.1/users.pdf

[Cohen] Cohen, "Advanced Topics in Computational Number Theory"

PYTHON MODULE INDEX

INDEX