Sage Reference Manual: Basic Structures

Release 6.6

The Sage Development Team

April 18, 2015

CONTENTS

CHAPTER

ONE

ABSTRACT BASE CLASS FOR SAGE OBJECTS

class sage.structure.sage_object.**SageObject**

Bases: object

Base class for all (user-visible) objects in Sage

Every object that can end up being returned to the user should inherit from SageObject.

_ascii_art_()

Return an ASCII art representation.

To implement multi-line ASCII art output in a derived class you must override this method. Unlike _repr_(), which is sometimes used for the hash key, the output of _ascii_art_() may depend on settings and is allowed to change during runtime.

OUTPUT:

An AsciiArt object, see sage.misc.ascii_art for details.

EXAMPLES:

You can use the ascii_art() function to get the ASCII art representation of any object in Sage:

```
sage: ascii_art(integral(exp(x+x^2)/(x+1), x))
  /
 |
 |   2
 |  x  + x
 | e
 | ------- dx
 |  x + 1
 |
/
```

Alternatively, you can use the %display ascii_art/simple magic to switch all output to ASCII art and back:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('tab = StandardTableaux(3)[2]; tab')
[[1, 2], [3]]
sage: shell.run_cell('%display ascii_art')
sage: shell.run_cell('tab')
1  2
3
sage: shell.run_cell('Tableaux.global_options(ascii_art="table", convention="French")')
sage: shell.run_cell('tab')
+---+
| 3 |
```

```
+---+---+
| 1 | 2 |
+---+---+
sage: shell.run_cell('%display plain')
sage: shell.run_cell('Tableaux.global_options.reset()')
sage: shell.quit()
```

TESTS:

```
sage: 1._ascii_art_()
1
sage: type(_)
<class 'sage.misc.ascii_art.AsciiArt'>
```

_cache_key()

Return a hashable key which identifies this objects for caching. The output must be hashable itself, or a tuple of objects which are hashable or define a `_cache_key`.

This method will only be called if the object itself is not hashable.

Some immutable objects (such as p-adic numbers) cannot implement a reasonable hash function because their == operator has been modified to return `True` for objects which might behave differently in some computations:

```
sage: K.<a> = Qq(9)
sage: b = a + O(3)
sage: c = a + 3
sage: b
a + O(3)
sage: c
a + 3 + O(3^20)
sage: b == c
True
sage: b == a
True
sage: c == a
False
```

If such objects defined a non-trivial hash function, this would break caching in many places. However, such objects should still be usable in caches. This can be achieved by defining an appropriate `_cache_key`:

```
sage: hash(b)
Traceback (most recent call last):
...
TypeError: unhashable type: 'sage.rings.padics.padic_ZZ_pX_CR_element.pAdicZZpXCRElement'
sage: @cached_method
....: def f(x): return x==a
sage: f(b)
True
sage: f(c) # if b and c were hashable, this would return True
False

sage: b._cache_key()
(..., ((0, 1),), 0, 1)
sage: c._cache_key()
(..., ((0, 1), (1,)), 0, 20)
```

An implementation must make sure that for elements `a` and `b`, if `a != b`, then also `a._cache_key() != b._cache_key()`. In practice this means that the `_cache_key` should always include the parent as its first argument:

```
sage: S.<a> = Qq(4)
sage: d = a + O(2)
sage: b._cache_key() == d._cache_key() # this would be True if the parents were not included
False
```

category()

db(*name*, *compress=True*)

Dumps self into the Sage database. Use db(name) by itself to reload.

The database directory is `$HOME/.sage/db`

TESTS:

```
sage: SageObject().db("Test")
doctest:... DeprecationWarning: db() is deprecated.
See http://trac.sagemath.org/2536 for details.
```

dump(*filename*, *compress=True*)

Same as self.save(filename, compress)

dumps(*compress=True*)

Dump `self` to a string `s`, which can later be reconstituted as `self` using `loads(s)`.

There is an optional boolean argument `compress` which defaults to `True`.

EXAMPLES:

```
sage: O=SageObject(); O.dumps()
'x\x9ck`J.NLO\xd5+.)*M.)-\x02\xb2\x80\xdc\xf8\xfc\xa4\xac\xd4\xe4\x12\xae` \xdb\x1f\xc2,d\xc
sage: O.dumps(compress=False)
'\x80\x02csage.structure.sage_object\nSageObject\nq\x01)\x81q\x02.'
```

parent()

Return the type of `self` to support the coercion framework.

EXAMPLES:

```
sage: t = log(sqrt(2) - 1) + log(sqrt(2) + 1); t
log(sqrt(2) + 1) + log(sqrt(2) - 1)
sage: u = t.maxima_methods()
sage: u.parent()
<class 'sage.symbolic.maxima_wrapper.MaximaWrapper'>
```

rename(*x=None*)

Change self so it prints as x, where x is a string.

Note: This is *only* supported for Python classes that derive from SageObject.

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x', sparse=True).gen()
sage: g = x^3 + x - 5
sage: g
x^3 + x - 5
sage: g.rename('a polynomial')
sage: g
a polynomial
sage: g + x
x^3 + 2*x - 5
```

```
sage: h = g^100
sage: str(h)[:20]
'x^300 + 100*x^298 - '
sage: h.rename('x^300 + ...')
sage: h
x^300 + ...
```

Real numbers are not Python classes, so rename is not supported:

```
sage: a = 3.14
sage: type(a)
<type 'sage.rings.real_mpfr.RealLiteral'>
sage: a.rename('pi')
Traceback (most recent call last):
...
NotImplementedError: object does not support renaming: 3.14000000000000
```

Note: The reason C-extension types are not supported by default is if they were then every single one would have to carry around an extra attribute, which would be slower and waste a lot of memory.

To support them for a specific class, add a `cdef public __custom_name` attribute.

---

reset_name()

Remove the custom name of an object.

EXAMPLES:

```
sage: P.<x> = QQ[]
sage: P
Univariate Polynomial Ring in x over Rational Field
sage: P.rename('A polynomial ring')
sage: P
A polynomial ring
sage: P.reset_name()
sage: P
Univariate Polynomial Ring in x over Rational Field
```

save(*filename=None*, *compress=True*)

Save self to the given filename.

EXAMPLES:

```
sage: f = x^3 + 5
sage: f.save(os.path.join(SAGE_TMP, 'file'))
sage: load(os.path.join(SAGE_TMP, 'file.sobj'))
x^3 + 5
```

version()

The version of Sage.

Call this to save the version of Sage in this object. If you then save and load this object it will know in what version of Sage it was created.

This only works on Python classes that derive from SageObject.

TESTS:

```
sage: v = DiGraph().version()
doctest:... DeprecationWarning: version() is deprecated.
See http://trac.sagemath.org/2536 for details.
```

sage.structure.sage_object.**dumps**(*obj*, *compress=True*)

Dump obj to a string s. To recover obj, use `loads(s)`.

See also:

`dumps()`

EXAMPLES:

```
sage: a = 2/3
sage: s = dumps(a)
sage: print len(s)
49
sage: loads(s)
2/3
```

sage.structure.sage_object.**load**(*compress=True*, *verbose=True*, **filename*)

Load Sage object from the file with name filename, which will have an `.sobj` extension added if it doesn't have one. Or, if the input is a filename ending in `.py`, `.pyx`, `.sage`, `.spyx`, `.f`, `.f90` or `.m`, load that file into the current running session.

Loaded files are not loaded into their own namespace, i.e., this is much more like Python's `execfile` than Python's `import`.

This function also loads a `.sobj` file over a network by specifying the full URL. (Setting `verbose = False` suppresses the loading progress indicator.)

Finally, if you give multiple positional input arguments, then all of those files are loaded, or all of the objects are loaded and a list of the corresponding loaded objects is returned.

EXAMPLE:

```
sage: u = 'http://sage.math.washington.edu/home/was/db/test.sobj'
sage: s = load(u)                                                  # optional - internet
Attempting to load remote file: http://sage.math.washington.edu/home/was/db/test.sobj
Loading: [.]
sage: s                                                            # optional - internet
'hello SAGE'
```

We test loading a file or multiple files or even mixing loading files and objects:

```
sage: t = tmp_filename(ext='.py')
sage: open(t,'w').write("print 'hello world'")
sage: load(t)
hello world
sage: load(t,t)
hello world
hello world
sage: t2 = tmp_filename(); save(2/3,t2)
sage: load(t,t,t2)
hello world
hello world
[None, None, 2/3]
```

We can load Fortran files:

```
sage: code = '      subroutine hello\n         print *, "Hello World!"\n      end subroutine hel
sage: t = tmp_filename(ext=".F")
sage: open(t, 'w').write(code)
sage: load(t)
sage: hello
<fortran object>
```

sage.structure.sage_object.**loads**(*s*, *compress=True*)

Recover an object x that has been dumped to a string s using `s = dumps(x)`.

See also:

`dumps()`

EXAMPLES:

```
sage: a = matrix(2, [1,2,3,-4/3])
sage: s = dumps(a)
sage: loads(s)
[   1    2]
[   3 -4/3]
```

If compress is True (the default), it will try to decompress the data with zlib and with bz2 (in turn); if neither succeeds, it will assume the data is actually uncompressed. If compress=False is explicitly specified, then no decompression is attempted.

```
sage: v = [1..10]
sage: loads(dumps(v, compress=False)) == v
True
sage: loads(dumps(v, compress=False), compress=True) == v
True
sage: loads(dumps(v, compress=True), compress=False)
Traceback (most recent call last):
...
UnpicklingError: invalid load key, 'x'.
```

sage.structure.sage_object.**picklejar**(*obj*, *dir=None*)

Create pickled sobj of obj in dir, with name the absolute value of the hash of the pickle of obj. This is used in conjunction with `unpickle_all()`.

To use this to test the whole Sage library right now, set the environment variable `SAGE_PICKLE_JAR`, which will make it so dumps will by default call picklejar with the default dir. Once you do that and doctest Sage, you'll find that the `SAGE_ROOT`/tmp/ contains a bunch of pickled objects along with corresponding txt descriptions of them. Use the `unpickle_all()` to see if they unpickle later.

INPUTS:

- `obj` – a pickleable object
- `dir` – a string or None; if None then `dir` defaults to `SAGE_ROOT/tmp/pickle_jar`

EXAMPLES:

```
sage: dir = tmp_dir()
sage: sage.structure.sage_object.picklejar(1, dir)
sage: sage.structure.sage_object.picklejar('test', dir)
sage: len(os.listdir(dir))   # Two entries (sobj and txt) for each object
4
```

TESTS:

Test an unaccessible directory:

```
sage: import os
sage: os.chmod(dir, 0o000)
sage: try:
...     uid = os.getuid()
... except AttributeError:
...    uid = -1
sage: if uid==0:
```

```
...     raise OSError('You must not run the doctests as root, geez!')
... else: sage.structure.sage_object.picklejar(1, dir + '/noaccess')
Traceback (most recent call last):
...
OSError: ...
sage: os.chmod(dir, 0o755)
```

sage.structure.sage_object.**register_unpickle_override**(*module*, *name*, *callable*, *call_name=None*)

Python pickles include the module and class name of classes. This means that rearranging the Sage source can invalidate old pickles. To keep the old pickles working, you can call register_unpickle_override with an old module name and class name, and the Python callable (function, class with __call__ method, etc.) to use for unpickling. (If this callable is a value in some module, you can specify the module name and class name, for the benefit of `explain_pickle()` when called with `in_current_sage=True`).)

EXAMPLES:

```
sage: from sage.structure.sage_object import unpickle_override, register_unpickle_override
sage: unpickle_global('sage.rings.integer', 'Integer')
<type 'sage.rings.integer.Integer'>
```

Now we horribly break the pickling system:

```
sage: register_unpickle_override('sage.rings.integer', 'Integer', Rational, call_name=('sage.rin
sage: unpickle_global('sage.rings.integer', 'Integer')
<type 'sage.rings.rational.Rational'>
```

and we reach into the internals and put it back:

```
sage: del unpickle_override[('sage.rings.integer', 'Integer')]
sage: unpickle_global('sage.rings.integer', 'Integer')
<type 'sage.rings.integer.Integer'>
```

In many cases, unpickling problems for old pickles can be resolved with a simple call to `register_unpickle_override`, as in the example above and in many of the `sage` source files. However, if the underlying data structure has changed significantly then unpickling may fail and it will be necessary to explicitly implement unpickling methods for the associated objects. The python pickle protocol is described in detail on the web and, in particular, in the python pickling documentation. For example, the following excerpt from this documentation shows that the unpickling of classes is controlled by their `__setstate__()` method.

```
object.__setstate__(state)

    Upon unpickling, if the class also defines the method :meth:`__setstate__`, it is
    called with the unpickled state. If there is no :meth:`__setstate__` method,
    the pickled state must be a dictionary and its items are assigned to the new
    instance's dictionary. If a class defines both :meth:`getstate__` and
    :meth:`__setstate__`, the state object needn't be a dictionary and these methods
    can do what they want.
```

By implementing a `__setstate__()` method for a class it should be possible to fix any unpickling problems for the class. As an example of what needs to be done, we show how to unpickle a `CombinatorialObject` object using a class which also inherits from `Element`. This exact problem often arises when refactoring old code into the element framework. First we create a pickle to play with:

```
sage: from sage.structure.element import Element
sage: class SourPickle(CombinatorialObject): pass
sage: class SweetPickle(CombinatorialObject,Element): pass
sage: import __main__
```

```
sage: __main__.SourPickle=SourPickle
sage: __main__.SweetPickle=SweetPickle  # a hack to allow us to pickle command line classes
sage: gherkin = dumps( SourPickle([1,2,3]) )
```

Using `register_unpickle_override()` we try to sweeten our pickle, but we are unable to eat it:

```
sage: from sage.structure.sage_object import register_unpickle_override
sage: register_unpickle_override('__main__','SourPickle',SweetPickle)
sage: loads( gherkin )
Traceback (most recent call last):
...
KeyError: 0
```

The problem is that the `SweetPickle` has inherited a `__setstate__()` method from `Element` which is not compatible with unpickling for `CombinatorialObject`. We can fix this by explicitly defining a new `__setstate__()` method:

```
sage: class SweeterPickle(CombinatorialObject,Element):
...       def __setstate__(self, state):
...           if isinstance(state, dict):   # a pickle from CombinatorialObject is just its inst
...               self._set_parent(Tableaux())      # this is a fudge: we need an appropriate pa
...               self.__dict__ = state
...           else:
...               self._set_parent(state[0])
...               self.__dict__ = state[1]
...
sage: __main__.SweeterPickle = SweeterPickle
sage: register_unpickle_override('__main__','SourPickle',SweeterPickle)
sage: loads( gherkin )
[1, 2, 3]
sage: loads(dumps( SweeterPickle([1,2,3]) ))   # check that pickles work for SweeterPickle
[1, 2, 3]
```

The `state` passed to `__setstate__()` will usually be something like the instance dictionary of the pickled object, however, with some older classes such as `CombinatorialObject` it will be a tuple. In general, the `state` can be any python object. `Sage` provides a special tool, `explain_pickle()`, which can help in figuring out the contents of an old pickle. Here is a second example.

```
sage: class A(object):
...      def __init__(self,value):
...          self.original_attribute = value
...      def __repr__(self):
...          return 'A(%s)'%self.original_attribute
sage: class B(object):
...      def __init__(self,value):
...          self.new_attribute = value
...      def __setstate__(self,state):
...          try:
...              self.new_attribute = state['new_attribute']
...          except KeyError:      # an old pickle
...              self.new_attribute = state['original_attribute']
...      def __repr__(self):
...          return 'B(%s)'%self.new_attribute
sage: import __main__
sage: __main__.A=A; __main__.B=B  # a hack to allow us to pickle command line classes
sage: A(10)
A(10)
sage: loads( dumps(A(10)) )
A(10)
```

```
sage: sage.misc.explain_pickle.explain_pickle( dumps(A(10)) )
pg_A = unpickle_global('__main__', 'A')
si = unpickle_newobj(pg_A, ())
pg_make_integer = unpickle_global('sage.rings.integer', 'make_integer')
unpickle_build(si, {'original_attribute':pg_make_integer('a')})
si
sage: from sage.structure.sage_object import register_unpickle_override
sage: register_unpickle_override('__main__', 'A', B)
sage: loads( dumps(A(10)) )
B(10)
sage: loads( dumps(B(10)) )
B(10)
```

Pickling for python classes and extension classes, such as cython, is different – again this is discussed in the python pickling documentation. For the unpickling of extension classes you need to write a `__reduce__()` method which typically returns a tuple `(f, args,...)` such that `f(*args)` returns (a copy of) the original object. The following code snippet is the `__reduce__()` method from `sage.rings.integer.Integer`.

```
def __reduce__(self):
    'Including the documentation properly causes a doc-test failure so we include it as a commen
    #* '''
    #* This is used when pickling integers.
    #*
    #* EXAMPLES::
    #*
    #*     sage: n = 5
    #*     sage: t = n.__reduce__(); t
    #*     (<built-in function make_integer>, ('5',))
    #*     sage: t[0](*t[1])
    #*     5
    #*     sage: loads(dumps(n)) == n
    #*     True
    #* '''
    # This single line below took me HOURS to figure out.
    # It is the *trick* needed to pickle Cython extension types.
    # The trick is that you must put a pure Python function
    # as the first argument, and that function must return
    # the result of unpickling with the argument in the second
    # tuple as input. All kinds of problems happen
    # if we don't do this.
    return sage.rings.integer.make_integer, (self.str(32),)
```

`sage.structure.sage_object.`**`save`**`(`*obj*, *filename=None*, *compress=True*, ***kwds*`)`

Save `obj` to the file with name `filename`, which will have an `.sobj` extension added if it doesn't have one and if `obj` doesn't have its own `save()` method, like e.g. Python tuples.

For image objects and the like (which have their own `save()` method), you may have to specify a specific extension, e.g. `.png`, if you don't want the object to be saved as a Sage object (or likewise, if `filename` could be interpreted as already having some extension).

> **Warning:** This will *replace* the contents of the file if it already exists.

EXAMPLES:

```
sage: a = matrix(2, [1,2,3,-5/2])
sage: objfile = os.path.join(SAGE_TMP, 'test.sobj')
sage: objfile_short = os.path.join(SAGE_TMP, 'test')
```

```
sage: save(a, objfile)
sage: load(objfile_short)
[   1    2]
[   3 -5/2]
sage: E = EllipticCurve([-1,0])
sage: P = plot(E)
sage: save(P, objfile_short)   # saves the plot to "test.sobj"
sage: save(P, filename=os.path.join(SAGE_TMP, "sage.png"), xmin=-2)
sage: save(P, os.path.join(SAGE_TMP, "filename.with.some.wrong.ext"))
Traceback (most recent call last):
...
ValueError: allowed file extensions for images are '.eps', '.pdf', '.png', '.ps', '.sobj', '.svg
sage: print load(objfile)
Graphics object consisting of 2 graphics primitives
sage: save("A python string", os.path.join(SAGE_TMP, 'test'))
sage: load(objfile)
'A python string'
sage: load(objfile_short)
'A python string'
```

TESTS:

Check that trac ticket #11577 is fixed:

```
sage: filename = os.path.join(SAGE_TMP, "foo.bar")  # filename containing a dot
sage: save((1,1),filename)             # saves tuple to "foo.bar.sobj"
sage: load(filename)
(1, 1)
```

`sage.structure.sage_object.`**`unpickle_all`**(*dir=None*, *debug=False*, *run_test_suite=False*)

Unpickle all sobj's in the given directory, reporting failures as they occur. Also printed the number of successes and failure.

INPUT:

- `dir` – a string; the name of a directory (or of a .tar.bz2 file that decompresses to a directory) full of pickles. (default: the standard pickle jar)
- `debug` – a boolean (default: False) whether to report a stacktrace in case of failure
- `run_test_suite` – a boolean (default: False) whether to run `TestSuite(x).run()` on the unpickled objects

EXAMPLES:

```
sage: dir = tmp_dir()
sage: sage.structure.sage_object.picklejar('hello', dir)
sage: sage.structure.sage_object.unpickle_all(dir)
Successfully unpickled 1 objects.
Failed to unpickle 0 objects.
```

When run with no arguments `unpickle_all()` tests that all of the "standard" pickles stored in the pickle_jar at `SAGE_ROOT/local/share/sage/ext/pickle_jar/pickle_jar.tar.bz2` can be unpickled.

```
sage: sage.structure.sage_object.unpickle_all()  # (4s on sage.math, 2011)
doctest:... DeprecationWarning: ...
See http://trac.sagemath.org/... for details.
Successfully unpickled ... objects.
Failed to unpickle 0 objects.
```

Check that unpickling a second time works (see trac ticket #5838):

```
sage: sage.structure.sage_object.unpickle_all()
Successfully unpickled ... objects.
Failed to unpickle 0 objects.
```

When it is not possible to unpickle a pickle in the pickle_jar then `unpickle_all()` prints the following error message which warns against removing pickles from the pickle_jar and directs the user towards `register_unpickle_override()`. The following code intentionally breaks a pickle to illustrate this:

```
sage: from sage.structure.sage_object import register_unpickle_override, unpickle_all, unpickle_
sage: class A(CombinatorialObject,sage.structure.element.Element):
...       pass # to break a pickle
sage: tableau_unpickler=unpickle_global('sage.combinat.tableau','Tableau_class')
sage: register_unpickle_override('sage.combinat.tableau','Tableau_class',A) # breaking the pickl
sage: unpickle_all()  # todo: not tested
...
Failed:
_class__sage_combinat_crystals_affine_AffineCrystalFromClassicalAndPromotion_with_category_eleme
_class__sage_combinat_crystals_tensor_product_CrystalOfTableaux_with_category_element_class__.sc
_class__sage_combinat_crystals_tensor_product_TensorProductOfCrystalsWithGenerators_with_categor
_class__sage_combinat_tableau_Tableau_class__.sobj
----------------------------------------------------------------------
** This error is probably due to an old pickle failing to unpickle.
** See sage.structure.sage_object.register_unpickle_override for
** how to override the default unpickling methods for (old) pickles.
** NOTE: pickles should never be removed from the pickle_jar!
----------------------------------------------------------------------
Successfully unpickled 583 objects.
Failed to unpickle 4 objects.
sage: register_unpickle_override('sage.combinat.tableau','Tableau_class',tableau_unpickler) # re
```

Todo

Create a custom-made `SourPickle` for the last example.

If you want to find *lots* of little issues in Sage then try the following:

```
sage: print "x"; sage.structure.sage_object.unpickle_all(run_test_suite = True) # todo: not test
```

This runs `TestSuite` tests on all objects in the Sage pickle jar. Some of those objects seem to unpickle properly, but do not pass the tests because their internal data structure is messed up. In most cases though it is just that their source file misses a TestSuite call, and therefore some misfeatures went unnoticed (typically Parents not implementing the `an_element` method).

Note: Every so often the standard pickle jar should be updated by running the doctest suite with the environment variable `SAGE_PICKLE_JAR` set, then copying the files from `SAGE_ROOT/tmp/pickle_jar*` into the standard pickle jar.

Warning: Sage's pickle jar helps to ensure backward compatibility in sage. Pickles should **only** be removed from the pickle jar after the corresponding objects have been properly deprecated. Any proposal to remove pickles from the pickle jar should first be discussed on sage-devel.

`sage.structure.sage_object.`**`unpickle_global`**(*module*, *name*)

Given a module name and a name within that module (typically a class name), retrieve the corresponding object. This normally just looks up the name in the module, but it can be overridden by register_unpickle_override. This is used in the Sage unpickling mechanism, so if the Sage source code organization changes, register_unpickle_override can allow old pickles to continue to work.

EXAMPLES:

```
sage: from sage.structure.sage_object import unpickle_override, register_unpickle_override
sage: unpickle_global('sage.rings.integer', 'Integer')
<type 'sage.rings.integer.Integer'>
```

Now we horribly break the pickling system:

```
sage: register_unpickle_override('sage.rings.integer', 'Integer', Rational, call_name=('sage.rin
sage: unpickle_global('sage.rings.integer', 'Integer')
<type 'sage.rings.rational.Rational'>
```

and we reach into the internals and put it back:

```
sage: del unpickle_override[('sage.rings.integer', 'Integer')]
sage: unpickle_global('sage.rings.integer', 'Integer')
<type 'sage.rings.integer.Integer'>
```

CHAPTER TWO

BASE CLASS FOR OBJECTS OF A CATEGORY

CLASS HIERARCHY:

- SageObject
  - **CategoryObject**
    - Parent

Many category objects in Sage are equipped with generators, which are usually special elements of the object. For example, the polynomial ring $\mathbf{Z}[x, y, z]$ is generated by x, y, and z. In Sage the `i` th generator of an object `X` is obtained using the notation `X.gen(i)`. From the Sage interactive prompt, the shorthand notation `X.i` is also allowed.

The following examples illustrate these functions in the context of multivariate polynomial rings and free modules.

EXAMPLES:

```
sage: R = PolynomialRing(ZZ, 3, 'x')
sage: R.ngens()
3
sage: R.gen(0)
x0
sage: R.gens()
(x0, x1, x2)
sage: R.variable_names()
('x0', 'x1', 'x2')
```

This example illustrates generators for a free module over $\mathbf{Z}$.

```
sage: M = FreeModule(ZZ, 4)
sage: M
Ambient free module of rank 4 over the principal ideal domain Integer Ring
sage: M.ngens()
4
sage: M.gen(0)
(1, 0, 0, 0)
sage: M.gens()
((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))
```

class `sage.structure.category_object.`**`CategoryObject`**

Bases: `sage.structure.sage_object.SageObject`

An object in some category.

`Hom`(*codomain*, *cat=None*)

Return the homspace `Hom(self, codomain, cat)` of all homomorphisms from self to codomain in the category cat. The default category is determined by `self.category()` and `codomain.category()`.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: R.Hom(QQ)
Set of Homomorphisms from Multivariate Polynomial Ring in x, y over Rational Field to Ration
```

Homspaces are defined for very general Sage objects, even elements of familiar rings.

```
sage: n = 5; Hom(n,7)
Set of Morphisms from 5 to 7 in Category of elements of Integer Ring
sage: z=(2/3); Hom(z,8/1)
Set of Morphisms from 2/3 to 8 in Category of elements of Rational Field
```

This example illustrates the optional third argument:

```
sage: QQ.Hom(ZZ, Sets())
Set of Morphisms from Rational Field to Integer Ring in Category of sets
```

base()

base_ring()

Return the base ring of `self`.

INPUT:

- `self` – an object over a base ring; typically a module

EXAMPLES:

```
sage: from sage.modules.module import Module
sage: Module(ZZ).base_ring()
Integer Ring

sage: F = FreeModule(ZZ,3)
sage: F.base_ring()
Integer Ring
sage: F.__class__.base_ring
<method 'base_ring' of 'sage.structure.category_object.CategoryObject' objects>
```

Note that the coordinates of the elements of a module can lie in a bigger ring, the `coordinate_ring`:

```
sage: M = (ZZ^2) * (1/2)
sage: v = M([1/2, 0])
sage: v.base_ring()
Integer Ring
sage: parent(v[0])
Rational Field
sage: v.coordinate_ring()
Rational Field
```

More examples:

```
sage: F = FreeAlgebra(QQ, 'x')
sage: F.base_ring()
Rational Field
sage: F.__class__.base_ring
<method 'base_ring' of 'sage.structure.category_object.CategoryObject' objects>

sage: E = CombinatorialFreeModule(ZZ, [1,2,3])
sage: F = CombinatorialFreeModule(ZZ, [2,3,4])
sage: H = Hom(E, F)
sage: H.base_ring()
```

```
Integer Ring
sage: H.__class__.base_ring
<method 'base_ring' of 'sage.structure.category_object.CategoryObject' objects>
```

Todo

Move this method elsewhere (typically in the Modules category) so as not to pollute the namespace of all category objects.

categories()

Return the categories of `self`.

EXAMPLES:

```
sage: ZZ.categories()
[Join of Category of euclidean domains
     and Category of infinite enumerated sets,
 Category of euclidean domains,
 Category of principal ideal domains,
 Category of unique factorization domains,
 Category of gcd domains,
 Category of integral domains,
 Category of domains,
 Category of commutative rings, ...
 Category of monoids, ...,
 Category of commutative additive groups, ...,
 Category of sets, ...,
 Category of objects]
```

category()

gens_dict()

Return a dictionary whose entries are `{var_name:variable,...}`.

gens_dict_recursive()

Return the dictionary of generators of `self` and its base rings.

OUTPUT:

- a dictionary with string names of generators as keys and generators of `self` and its base rings as values.

EXAMPLES:

```
sage: R = QQ['x,y']['z,w']
sage: sorted(R.gens_dict_recursive().items())
[('w', w), ('x', x), ('y', y), ('z', z)]
```

has_base(*category=None*)

inject_variables(*scope=None, verbose=True*)

Inject the generators of self with their names into the namespace of the Python code from which this function is called. Thus, e.g., if the generators of self are labeled 'a', 'b', and 'c', then after calling this method the variables a, b, and c in the current scope will be set equal to the generators of self.

NOTE: If Foo is a constructor for a Sage object with generators, and Foo is defined in Cython, then it would typically call `inject_variables()` on the object it creates. E.g., `PolynomialRing(QQ, 'y')` does this so that the variable y is the generator of the polynomial ring.

injvar (*scope=None*, *verbose=True*)
This is a deprecated synonym for `inject_variables()`.

latex_name ()

latex_variable_names ()
Returns the list of variable names suitable for latex output.

All `_SOMETHING` substrings are replaced by `_{SOMETHING}` recursively so that subscripts of subscripts work.

EXAMPLES:

```
sage: R, x = PolynomialRing(QQ,'x',12).objgens()
sage: x
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11)
sage: print R.latex_variable_names ()
['x_{0}', 'x_{1}', 'x_{2}', 'x_{3}', 'x_{4}', 'x_{5}', 'x_{6}', 'x_{7}', 'x_{8}', 'x_{9}', '
sage: f = x[0]^3 + 15/3 * x[1]^10
sage: print latex(f)
5 x_{1}^{10} + x_{0}^{3}
```

normalize_names (*ngens*, *names=None*)

objgen ()
Return the tuple `(self, self.gen())`.

EXAMPLES:

```
sage: R, x = PolynomialRing(QQ,'x').objgen()
sage: R
Univariate Polynomial Ring in x over Rational Field
sage: x
x
```

objgens ()
Return the tuple `(self, self.gens())`.

EXAMPLES:

```
sage: R = PolynomialRing(QQ, 3, 'x'); R
Multivariate Polynomial Ring in x0, x1, x2 over Rational Field
sage: R.objgens()
(Multivariate Polynomial Ring in x0, x1, x2 over Rational Field, (x0, x1, x2))
```

variable_name ()

variable_names ()

`sage.structure.category_object.`**check_default_category** (*default_category*, *category*)

`sage.structure.category_object.`**guess_category** (*obj*)

class `sage.structure.category_object.`**localvars** (*obj*, *names*, *latex_names=None*, *normalize=True*)
Context manager for safely temporarily changing the variables names of an object with generators.

Objects with named generators are globally unique in Sage. Sometimes, though, it is very useful to be able to temporarily display the generators differently. The new Python `with` statement and the localvars context manager make this easy and safe (and fun!)

Suppose X is any object with generators. Write

```
with localvars(X, names[, latex_names] [,normalize=False]):
    some code
    ...
```

and the indented code will be run as if the names in `X` are changed to the new names. If you give `normalize=True`, then the names are assumed to be a tuple of the correct number of strings.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: with localvars(R, 'z,w'):
...       print x^3 + y^3 - x*y
...
z^3 + w^3 - z*w
```

NOTES: I wrote this because it was needed to print elements of the quotient of a ring R by an ideal I using the print function for elements of R. See the code in `sage.rings.quotient_ring_element`.

AUTHOR: William Stein (2006-10-31)

CHAPTER

THREE

BASE CLASS FOR OLD-STYLE PARENT OBJECTS

CLASS HIERARCHY:

SageObject

Parent

ParentWithBase ParentWithGens

TESTS:

This came up in some subtle bug once.

```
sage: gp(2) + gap(3)
5
```

class sage.structure.parent_old.**Parent**

Bases: sage.structure.parent.Parent

Parents are the SAGE/mathematical analogues of container objects in computer science.

TESTS:

```
sage: V = VectorSpace(GF(2,'a'),2)
sage: V.list()
[(0, 0), (1, 0), (0, 1), (1, 1)]
sage: MatrixSpace(GF(3), 1, 1).list()
[[0], [1], [2]]
sage: DirichletGroup(3).list()
[Dirichlet character modulo 3 of conductor 1 mapping 2 |--> 1,
Dirichlet character modulo 3 of conductor 3 mapping 2 |--> -1]
sage: K = GF(7^6,'a')
sage: K.list()[:10] # long time
[0, 1, 2, 3, 4, 5, 6, a, a + 1, a + 2]
sage: K.<a> = GF(4)
sage: K.list()
[0, a, a + 1, 1]
```

coerce_map_from_c(*S*)

EXAMPLES:

Check to make sure that we handle coerce maps from Python native types correctly:

```
sage: QQ['q,t'].coerce_map_from(int)
Composite map:
  From: Set of Python objects of type 'int'
  To:   Multivariate Polynomial Ring in q, t over Rational Field
  Defn:   Native morphism:
          From: Set of Python objects of type 'int'
```

```
      To:   Rational Field
    then
      Polynomial base injection morphism:
      From: Rational Field
      To:   Multivariate Polynomial Ring in q, t over Rational Field
```

coerce_map_from_impl(*S*)

get_action_c(*S*, *op*, *self_on_left*)

get_action_impl(*S*, *op*, *self_on_left*)

has_coerce_map_from_c(*S*)
Return True if there is a natural map from S to self. Otherwise, return False.

has_coerce_map_from_impl(*S*)

list()
Return a list of the elements of `self`.

OUTPUT:

A list of all the elements produced by the iterator defined for the object. The result is cached. An infinite set may define an iterator, allowing one to search through the elements, but a request by this method for the entire list should fail.

NOTE:

Some objects `X` do not know if they are finite or not. If `X.is_finite()` fails with a `NotImplementedError`, then `X.list()` will simply try. In that case, it may run without stopping.

However, if `X` knows that it is infinite, then running `X.list()` will raise an appropriate error, while running `list(X)` will run indefinitely. For many Sage objects `X`, using `X.list()` is preferable to using `list(X)`.

Nevertheless, since the whole list of elements is created and cached by `X.list()`, it may be better to do `for x in X:`, not `for x in X.list():`.

EXAMPLES:

```
sage: R = Integers(11)
sage: R.list()    # indirect doctest
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sage: ZZ.list()
Traceback (most recent call last):
...
NotImplementedError: since it is infinite, cannot list Integer Ring
```

This is the motivation for trac ticket #10470

```
sage: (QQ^2).list()
Traceback (most recent call last):
...
NotImplementedError: since it is infinite, cannot list Vector space of dimension 2 over Rati
```

TESTS:

The following tests the caching by adjusting the cached version:

```
sage: R = Integers(3)
sage: R.list()
[0, 1, 2]
```

```
sage: R._list[0] = 'junk'
sage: R.list()
['junk', 1, 2]
```

Here we test that for an object that does not know whether it is finite or not. Calling `X.list()` simply tries to create the list (but here it fails, since the object is not iterable). This was fixed trac ticket #11350

```
sage: R.<t,p> = QQ[]
sage: Q = R.quotient(t^2-t+1)
sage: Q.is_finite()
Traceback (most recent call last):
...
NotImplementedError
sage: Q.list()
Traceback (most recent call last):
...
NotImplementedError: object does not support iteration
```

Here is another example. We artificially create a version of the ring of integers that does not know whether it is finite or not:

```
sage: from sage.rings.integer_ring import IntegerRing_class
sage: class MyIntegers_class(IntegerRing_class):
....:     def is_finite(self):
....:         raise NotImplementedError
sage: MyIntegers = MyIntegers_class()
sage: MyIntegers.is_finite()
Traceback (most recent call last):
...
NotImplementedError
```

Asking for `list(MyIntegers)` below will never finish without pressing Ctrl-C. We let it run for 1 second and then interrupt:

```
sage: alarm(1.0); list(MyIntegers)
Traceback (most recent call last):
...
AlarmInterrupt
```

CHAPTER

FOUR

BASE CLASS FOR OLD-STYLE PARENT OBJECTS WITH A BASE RING

class sage.structure.parent_base.**ParentWithBase**

Bases: sage.structure.parent_old.Parent

This class is being deprecated, see parent.Parent for the new model.

base_extend(*X*)

sage.structure.parent_base.**is_ParentWithBase**(*x*)

Return True if x is a parent object with base.

CHAPTER

FIVE

BASE CLASS FOR OLD-STYLE PARENT OBJECTS WITH GENERATORS

Note: This class is being deprecated, see `sage.structure.parent.Parent` and `sage.structure.category_object.CategoryObject` for the new model.

Many parent objects in Sage are equipped with generators, which are special elements of the object. For example, the polynomial ring $\mathbf{Z}[x, y, z]$ is generated by x, y, and z. In Sage the i^{th} generator of an object `X` is obtained using the notation `X.gen(i)`. From the Sage interactive prompt, the shorthand notation `X.i` is also allowed.

REQUIRED: A class that derives from ParentWithGens *must* define the ngens() and gen(i) methods.

OPTIONAL: It is also good if they define gens() to return all gens, but this is not necessary.

The `gens` function returns a tuple of all generators, the `ngens` function returns the number of generators.

The `_assign_names` functions is for internal use only, and is called when objects are created to set the generator names. It can only be called once.

The following examples illustrate these functions in the context of multivariate polynomial rings and free modules.

EXAMPLES:

```
sage: R = PolynomialRing(ZZ, 3, 'x')
sage: R.ngens()
3
sage: R.gen(0)
x0
sage: R.gens()
(x0, x1, x2)
sage: R.variable_names()
('x0', 'x1', 'x2')
```

This example illustrates generators for a free module over $\mathbf{Z}$.

```
sage: M = FreeModule(ZZ, 4)
sage: M
Ambient free module of rank 4 over the principal ideal domain Integer Ring
sage: M.ngens()
4
sage: M.gen(0)
(1, 0, 0, 0)
sage: M.gens()
((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))
```

class `sage.structure.parent_gens.`**`ParentWithAdditiveAbelianGens`**

Bases: `sage.structure.parent_gens.ParentWithGens`

EXAMPLES:

```
sage: class MyParent(ParentWithGens):
...       def ngens(self): return 3
sage: P = MyParent(base = QQ, names = 'a,b,c', normalize = True, category = Groups())
sage: P.category()
Category of groups
sage: P._names
('a', 'b', 'c')
```

generator_orders()

class sage.structure.parent_gens.**ParentWithGens**

Bases: sage.structure.parent_base.ParentWithBase

EXAMPLES:

```
sage: class MyParent(ParentWithGens):
...       def ngens(self): return 3
sage: P = MyParent(base = QQ, names = 'a,b,c', normalize = True, category = Groups())
sage: P.category()
Category of groups
sage: P._names
('a', 'b', 'c')
```

gen(*i=0*)

gens()

Return a tuple whose entries are the generators for this object, in order.

hom(*im_gens*, *codomain=None*, *check=True*)

Return the unique homomorphism from self to codomain that sends `self.gens()` to the entries of `im_gens`. Raises a TypeError if there is no such homomorphism.

INPUT:

- `im_gens` - the images in the codomain of the generators of this object under the homomorphism
- `codomain` - the codomain of the homomorphism
- `check` - whether to verify that the images of generators extend to define a map (using only canonical coercions).

OUTPUT:

- a homomorphism self –> codomain

Note: As a shortcut, one can also give an object X instead of `im_gens`, in which case return the (if it exists) natural map to X.

EXAMPLE: Polynomial Ring We first illustrate construction of a few homomorphisms involving a polynomial ring.

```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = R.hom([5], QQ)
sage: f(x^2 - 19)
6

sage: R.<x> = PolynomialRing(QQ)
sage: f = R.hom([5], GF(7))
Traceback (most recent call last):
...
TypeError: images do not define a valid homomorphism
```

```
sage: R.<x> = PolynomialRing(GF(7))
sage: f = R.hom([3], GF(49,'a'))
sage: f
Ring morphism:
  From: Univariate Polynomial Ring in x over Finite Field of size 7
  To:   Finite Field in a of size 7^2
  Defn: x |--> 3
sage: f(x+6)
2
sage: f(x^2+1)
3
```

EXAMPLE: Natural morphism

```
sage: f = ZZ.hom(GF(5))
sage: f(7)
2
sage: f
Ring Coercion morphism:
  From: Integer Ring
  To:   Finite Field of size 5
```

There might not be a natural morphism, in which case a TypeError exception is raised.

```
sage: QQ.hom(ZZ)
Traceback (most recent call last):
...
TypeError: Natural coercion morphism from Rational Field to Integer Ring not defined.
```

ngens()

class sage.structure.parent_gens.**ParentWithMultiplicativeAbelianGens**

Bases: sage.structure.parent_gens.ParentWithGens

EXAMPLES:

```
sage: class MyParent(ParentWithGens):
...       def ngens(self): return 3
sage: P = MyParent(base = QQ, names = 'a,b,c', normalize = True, category = Groups())
sage: P.category()
Category of groups
sage: P._names
('a', 'b', 'c')
```

generator_orders()

sage.structure.parent_gens.**is_ParentWithAdditiveAbelianGens**(*x*)

Return True if x is a parent object with additive abelian generators, i.e., derives from sage.structure.parent_gens.ParentWithAdditiveAbelianGens and False otherwise.

EXAMPLES:

```
sage: from sage.structure.parent_gens import is_ParentWithAdditiveAbelianGens
sage: is_ParentWithAdditiveAbelianGens(QQ)
False
sage: is_ParentWithAdditiveAbelianGens(QQ^3)
True
```

sage.structure.parent_gens.**is_ParentWithGens**(*x*)

Return True if x is a parent object with generators, i.e., derives from sage.structure.parent_gens.ParentWithGens and False otherwise.

EXAMPLES:

```
sage: from sage.structure.parent_gens import is_ParentWithGens
sage: is_ParentWithGens(QQ['x'])
True
sage: is_ParentWithGens(CC)
True
sage: is_ParentWithGens(Primes())
False
```

sage.structure.parent_gens.**is_ParentWithMultiplicativeAbelianGens**(*x*)

Return True if x is a parent object with additive abelian generators, i.e., derives from sage.structure.parent_gens.ParentWithMultiplicativeAbelianGens and False otherwise.

EXAMPLES:

```
sage: from sage.structure.parent_gens import is_ParentWithMultiplicativeAbelianGens
sage: is_ParentWithMultiplicativeAbelianGens(QQ)
False
sage: is_ParentWithMultiplicativeAbelianGens(DirichletGroup(11))
True
```

class sage.structure.parent_gens.**localvars**

Bases: object

Context manager for safely temporarily changing the variables names of an object with generators.

Objects with named generators are globally unique in Sage. Sometimes, though, it is very useful to be able to temporarily display the generators differently. The new Python with statement and the localvars context manager make this easy and safe (and fun!)

Suppose X is any object with generators. Write

```
with localvars(X, names[, latex_names] [,normalize=False]):
     some code
     ...
```

and the indented code will be run as if the names in X are changed to the new names. If you give normalize=True, then the names are assumed to be a tuple of the correct number of strings.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: with localvars(R, 'z,w'):
...       print x^3 + y^3 - x*y
...
z^3 + w^3 - z*w
```

Note: I wrote this because it was needed to print elements of the quotient of a ring R by an ideal I using the print function for elements of R. See the code in quotient_ring_element.pyx.

AUTHOR:

- William Stein (2006-10-31)

sage.structure.parent_gens.**normalize_names**(*ngens*, *names=None*)

Return a tuple of strings of variable names of length ngens given the input names.

INPUT:

- ngens - integer
- names
  - tuple or list of strings, such as ('x', 'y')
  - a string prefix, such as 'alpha'
  - string of single character names, such as 'xyz'

EXAMPLES:

```
sage: from sage.structure.parent_gens import normalize_names as nn
sage: nn(1, 'a')
('a',)
sage: nn(2, 'zzz')
('zzz0', 'zzz1')
sage: nn(2, 'ab')
('a', 'b')
sage: nn(3, ('a', 'bb', 'ccc'))
('a', 'bb', 'ccc')
sage: nn(4, ['a1', 'a2', 'b1', 'b11'])
('a1', 'a2', 'b1', 'b11')
```

TESTS:

```
sage: nn(2, 'z1')
('z10', 'z11')
sage: PolynomialRing(QQ, 2, 'alpha0')
Multivariate Polynomial Ring in alpha00, alpha01 over Rational Field
```

CHAPTER

SIX

CONTAINERS FOR STORING COERCION DATA

This module provides `TripleDict` and `MonoDict`. These are structures similar to `WeakKeyDictionary` in Python's weakref module, and are optimized for lookup speed. The keys for `TripleDict` consist of triples (k1,k2,k3) and are looked up by identity rather than equality. The keys are stored by weakrefs if possible. If any one of the components k1, k2, k3 gets garbage collected, then the entry is removed from the `TripleDict`.

Key components that do not allow for weakrefs are stored via a normal refcounted reference. That means that any entry stored using a triple (k1,k2,k3) so that none of the k1,k2,k3 allows a weak reference behaves as an entry in a normal dictionary: Its existence in `TripleDict` prevents it from being garbage collected.

That container currently is used to store coercion and conversion maps between two parents (trac ticket #715) and to store homsets of pairs of objects of a category (trac ticket #11521). In both cases, it is essential that the parent structures remain garbage collectable, it is essential that the data access is faster than with a usual `WeakKeyDictionary`, and we enforce the "unique parent condition" in Sage (parent structures should be identical if they are equal).

`MonoDict` behaves similarly, but it takes a single item as a key. It is used for caching the parents which allow a coercion map into a fixed other parent (trac ticket #12313).

By trac ticket #14159, `MonoDict` and `TripleDict` can be optionally used with weak references on the values.

class `sage.structure.coerce_dict.`**`MonoDict`**

Bases: `object`

This is a hashtable specifically designed for (read) speed in the coercion model.

It differs from a python WeakKeyDictionary in the following important ways:

- Comparison is done using the 'is' rather than '==' operator.
- Only weak references to the keys are stored if at all possible. Keys that do not allow for weak references are stored with a normal refcounted reference.
- The callback of the weak references is safe against recursion, see below.

There are special cdef set/get methods for faster access. It is bare-bones in the sense that not all dictionary methods are implemented.

IMPLEMENTATION:

It is implemented as a hash table with open addressing, similar to python's dict.

If ki supports weak references then ri is a weak reference to ki with a callback to remove the entry from the dictionary if ki gets garbage collected. If ki is does not support weak references then ri is identical to ki. In the latter case the presence of the key in the dictionary prevents it from being garbage collected.

INPUT:

- `size` – unused parameter, present for backward compatibility.

- `data` – optional iterable defining initial data.
- `threshold` – unused parameter, present for backward compatibility.
- `weak_values` – optional bool (default False). If it is true, weak references to the values in this dictionary will be used, when possible.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import MonoDict
sage: L = MonoDict()
sage: a = 'a'; b = 'ab'; c = '-15'
sage: L[a] = 1
sage: L[b] = 2
sage: L[c] = 3
```

The key is expected to be a unique object. Hence, the item stored for `c` can not be obtained by providing another equal string:

```
sage: L[a]
1
sage: L[b]
2
sage: L[c]
3
sage: L['-15']
Traceback (most recent call last):
...
KeyError: '-15'
```

Not all features of Python dictionaries are available, but iteration over the dictionary items is possible:

```
sage: # for some reason the following failed in "make ptest"
sage: # on some installations, see #12313 for details
sage: sorted(L.iteritems()) # random layout
[('-15', 3), ('a', 1), ('ab', 2)]
sage: # the following seems to be more consistent
sage: set(L.iteritems())
{('-15', 3), ('a', 1), ('ab', 2)}
sage: del L[c]
sage: sorted(L.iteritems())
[('a', 1), ('ab', 2)]
sage: len(L)
2
sage: for i in range(1000):
...       L[i] = i
sage: len(L)
1002
sage: L['a']
1
sage: L['c']
Traceback (most recent call last):
...
KeyError: 'c'
```

Note that this kind of dictionary is also used for caching actions and coerce maps. In previous versions of Sage, the cache was by strong references and resulted in a memory leak in the following example. However, this leak was fixed by trac ticket #715, using weak references:

```
sage: K = GF(1<<55,'t')
sage: for i in range(50):
...     a = K.random_element()
...     E = EllipticCurve(j=a)
...     P = E.random_point()
...     Q = 2*P
sage: import gc
sage: n = gc.collect()
sage: from sage.schemes.elliptic_curves.ell_finite_field import EllipticCurve_finite_field
sage: LE = [x for x in gc.get_objects() if isinstance(x, EllipticCurve_finite_field)]
sage: len(LE)    # indirect doctest
1
```

TESTS:

Here, we demonstrate the use of weak values.

```
sage: M = MonoDict(13)
sage: MW = MonoDict(13, weak_values=True)
sage: class Foo: pass
sage: a = Foo()
sage: b = Foo()
sage: k = 1
sage: M[k] = a
sage: MW[k] = b
sage: M[k] is a
True
sage: MW[k] is b
True
sage: k in M
True
sage: k in MW
True
```

While `M` uses a strong reference to `a`, `MW` uses a *weak* reference to `b`, and after deleting `b`, the corresponding item of `MW` will be removed during the next garbage collection:

```
sage: import gc
sage: del a,b
sage: _ = gc.collect()
sage: k in M
True
sage: k in MW
False
sage: len(MW)
0
sage: len(M)
1
```

Note that `MW` also accepts values that do not allow for weak references:

```
    sage: MW[k] = int(5)
    sage: MW[k]
    5

The following demonstrates that :class:`MonoDict` is safer than
:class:`~weakref.WeakKeyDictionary` against recursions created by nested
callbacks; compare :trac:`15069` (the mechanism used now is different, though)::

    sage: M = MonoDict(11)
    sage: class A: pass
    sage: a = A()
    sage: prev = a
    sage: for i in range(1000):
    ....:     newA = A()
    ....:     M[prev] = newA
    ....:     prev = newA
    sage: len(M)
    1000
    sage: del a
    sage: len(M)
    0

The corresponding example with a Python :class:`weakref.WeakKeyDictionary`
would result in a too deep recursion during deletion of the dictionary
items::

    sage: import weakref
    sage: M = weakref.WeakKeyDictionary()
    sage: a = A()
    sage: prev = a
    sage: for i in range(1000):
    ....:     newA = A()
    ....:     M[prev] = newA
    ....:     prev = newA
    sage: len(M)
    1000
    sage: del a
    Exception RuntimeError: 'maximum recursion depth exceeded while calling a Python object' in
    sage: len(M)>0
    True

Check that also in the presence of circular references, :class:`MonoDict`
gets properly collected::

    sage: import gc
    sage: def count_type(T):
    ....:     return len([c for c in gc.get_objects() if isinstance(c,T)])
    sage: _=gc.collect()
    sage: N=count_type(MonoDict)
    sage: for i in range(100):
    ....:     V = [ MonoDict(11,{"id":j+100*i}) for j in range(100)]
    ....:     n= len(V)
    ....:     for i in range(n): V[i][V[(i+1)%n]]=(i+1)%n
    ....:     del V
    ....:     _=gc.collect()
    ....:     assert count_type(MonoDict) == N
    sage: count_type(MonoDict) == N
    True
```

```
AUTHORS:

- Simon King (2012-01)
- Nils Bruin (2012-08)
- Simon King (2013-02)
- Nils Bruin (2013-11)
```

iteritems()

EXAMPLES:

```
sage: from sage.structure.coerce_dict import MonoDict
sage: L = MonoDict(31)
sage: L[1] = None
sage: L[2] = True
sage: list(sorted(L.iteritems()))
[(1, None), (2, True)]
```

class sage.structure.coerce_dict.**MonoDictEraser**

Bases: object

Erase items from a MonoDict when a weak reference becomes invalid.

This is of internal use only. Instances of this class will be passed as a callback function when creating a weak reference.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import MonoDict
sage: class A: pass
sage: a = A()
sage: M = MonoDict()
sage: M[a] = 1
sage: len(M)
1
sage: del a
sage: import gc
sage: n = gc.collect()
sage: len(M)    # indirect doctest
0
```

AUTHOR:

- Simon King (2012-01)
- Nils Bruin (2013-11)

class sage.structure.coerce_dict.**TripleDict**

Bases: object

This is a hashtable specifically designed for (read) speed in the coercion model.

It differs from a python dict in the following important ways:

- All keys must be sequence of exactly three elements. All sequence types (tuple, list, etc.) map to the same item.
- Comparison is done using the 'is' rather than '==' operator.

There are special cdef set/get methods for faster access. It is bare-bones in the sense that not all dictionary methods are implemented.

It is implemented as a list of lists (hereafter called buckets). The bucket is chosen according to a very simple hash based on the object pointer, and each bucket is of the form [id(k1), id(k2), id(k3), r1, r2, r3, value, id(k1), id(k2), id(k3), r1, r2, r3, value, ...], on which a linear search is performed. If a key component ki supports weak references then ri is a weak reference to ki; otherwise ri is identical to ki.

INPUT:

- `size` – an integer, the initial number of buckets. To spread objects evenly, the size should ideally be a prime, and certainly not divisible by 2.
- `data` – optional iterable defining initial data.
- `threshold` – optional number, default 0.7. It determines how frequently the dictionary will be resized (large threshold implies rare resizing).
- `weak_values` – optional bool (default False). If it is true, weak references to the values in this dictionary will be used, when possible.

If any of the key components k1,k2,k3 (this can happen for a key component that supports weak references) gets garbage collected then the entire entry disappears. In that sense this structure behaves like a nested `WeakKeyDictionary`.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import TripleDict
sage: L = TripleDict()
sage: a = 'a'; b = 'b'; c = 'c'
sage: L[a,b,c] = 1
sage: L[a,b,c]
1
sage: L[c,b,a] = -1
sage: list(L.iteritems())      # random order of output.
[(('c', 'b', 'a'), -1), (('a', 'b', 'c'), 1)]
sage: del L[a,b,c]
sage: list(L.iteritems())
[(('c', 'b', 'a'), -1)]
sage: len(L)
1
sage: for i in range(1000):
...       L[i,i,i] = i
sage: len(L)
1001
sage: L = TripleDict(L)
sage: L[c,b,a]
-1
sage: L[a,b,c]
Traceback (most recent call last):
...
KeyError: ('a', 'b', 'c')
sage: L[a]
Traceback (most recent call last):
...
KeyError: 'a'
sage: L[a] = 1
Traceback (most recent call last):
...
KeyError: 'a'
```

Note that this kind of dictionary is also used for caching actions and coerce maps. In previous versions of Sage, the cache was by strong references and resulted in a memory leak in the following example. However, this leak was fixed by trac ticket #715, using weak references:

```
sage: K = GF(1<<55,'t')
sage: for i in range(50):
...     a = K.random_element()
...     E = EllipticCurve(j=a)
...     P = E.random_point()
...     Q = 2*P
sage: import gc
sage: n = gc.collect()
sage: from sage.schemes.elliptic_curves.ell_finite_field import EllipticCurve_finite_field
sage: LE = [x for x in gc.get_objects() if isinstance(x, EllipticCurve_finite_field)]
sage: len(LE)    # indirect doctest
1
```

TESTS:

Here, we demonstrate the use of weak values.

```
sage: class Foo: pass
sage: T = TripleDict(13)
sage: TW = TripleDict(13, weak_values=True)
sage: a = Foo()
sage: b = Foo()
sage: k = 1
sage: T[a,k,k]=1
sage: T[k,a,k]=2
sage: T[k,k,a]=3
sage: T[k,k,k]=a
sage: TW[b,k,k]=1
sage: TW[k,b,k]=2
sage: TW[k,k,b]=3
sage: TW[k,k,k]=b
sage: len(T)
4
sage: len(TW)
4
sage: (k,k,k) in T
True
sage: (k,k,k) in TW
True
sage: T[k,k,k] is a
True
sage: TW[k,k,k] is b
True
```

Now, `T` holds a strong reference to `a`, namely in `T[k,k,k]`. Hence, when we delete `a`, *all* items of `T` survive:

```
sage: del a
sage: _ = gc.collect()
sage: len(T)
4
```

Only when we remove the strong reference, the items become collectable:

```
sage: del T[k,k,k]
sage: _ = gc.collect()
sage: len(T)
0
```

The situation is different for `TW`, since it only holds *weak* references to `a`. Therefore, all items become collectable after deleting `a`:

```
sage: del b
sage: _ = gc.collect()
sage: len(TW)
0
```

Note: The index h corresponding to the key [k1, k2, k3] is computed as a value of unsigned type size_t as follows:

$$h = id(k1) + 13 * id(k2) xor 503 id(k3)$$

The natural type for this quantity is Py_ssize_t, which is a signed quantity with the same length as size_t. Storing it in a signed way gives the most efficient storage into PyInt, while preserving sign information.

In previous situations there were some problems with ending up with negative indices, which required casting to an unsigned type, i.e., (<size_t> h)% N since C has a sign-preserving % operation This caused problems on 32 bits systems, see trac ticket #715 for details. This is irrelevant for the current implementation.

AUTHORS:

- Robert Bradshaw, 2007-08
- Simon King, 2012-01
- Nils Bruin, 2012-08
- Simon King, 2013-02
- Nils Bruin, 2013-11

iteritems()

EXAMPLES:

```
sage: from sage.structure.coerce_dict import TripleDict
sage: L = TripleDict(31)
sage: L[1,2,3] = None
sage: list(L.iteritems())
[((1, 2, 3), None)]
```

class sage.structure.coerce_dict.**TripleDictEraser**

Bases: object

Erases items from a TripleDict when a weak reference becomes invalid.

This is of internal use only. Instances of this class will be passed as a callback function when creating a weak reference.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import TripleDict
sage: class A: pass
sage: a = A()
sage: T = TripleDict()
sage: T[a,ZZ,None] = 1
sage: T[ZZ,a,1] = 2
sage: T[a,a,ZZ] = 3
sage: len(T)
3
sage: del a
sage: import gc
sage: n = gc.collect()
sage: len(T) # indirect doctest
0
```

AUTHOR:

- Simon King (2012-01)
- Nils Bruin (2013-11)

`sage.structure.coerce_dict.signed_id(x)`

A function like Python's `id()` returning *signed* integers, which are guaranteed to fit in a `Py_ssize_t`.

Theoretically, there is no guarantee that two different Python objects have different `signed_id()` values. However, under the mild assumption that a C pointer fits in a `Py_ssize_t`, this is guaranteed.

TESTS:

```
sage: a = 1.23e45  # some object
sage: from sage.structure.coerce_dict import signed_id
sage: s = signed_id(a)
sage: id(a) == s or id(a) == s + 2**32 or id(a) == s + 2**64
True
sage: signed_id(a) <= sys.maxsize
True
```

CHAPTER

SEVEN

FORMAL SUMS

AUTHORS:

- David Harvey (2006-09-20): changed FormalSum not to derive from "list" anymore, because that breaks new Element interface
- Nick Alexander (2006-12-06): added test cases.
- William Stein (2006, 2009): wrote the first version in 2006, documented it in 2009.
- Volker Braun (2010-07-19): new-style coercions, documentation added. FormalSums now derives from UniqueRepresentation.

FUNCTIONS:

- **`FormalSums(ring)` – create the module of formal finite sums with** coefficients in the given ring.
- `FormalSum(list of pairs (coeff, number))` – create a formal sum

EXAMPLES:

```
sage: A = FormalSum([(1, 2/3)]); A
2/3
sage: B = FormalSum([(3, 1/5)]); B
3*1/5
sage: -B
-3*1/5
sage: A + B
3*1/5 + 2/3
sage: A - B
-3*1/5 + 2/3
sage: B*3
9*1/5
sage: 2*A
2*2/3
sage: list(2*A + A)
[(3, 2/3)]
```

TESTS:

```
sage: R = FormalSums(QQ)
sage: loads(dumps(R)) == R
True
sage: a = R(2/3) + R(-5/7); a
-5/7 + 2/3
sage: loads(dumps(a)) == a
True
```

class sage.structure.formal_sum.**FormalSum**(*x*, *parent=Abelian Group of all Formal Finite Sums over Integer Ring*, *check=True*, *reduce=True*)

Bases: sage.structure.element.ModuleElement

A formal sum over a ring.

reduce()

EXAMPLES:

```
sage: a = FormalSum([(-2,3), (2,3)], reduce=False); a
-2*3 + 2*3
sage: a.reduce()
sage: a
0
```

class sage.structure.formal_sum.**FormalSums**(*base_ring*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.modules.module.Module_old

The R-module of finite formal sums with coefficients in some ring R.

EXAMPLES:

```
sage: FormalSums()
Abelian Group of all Formal Finite Sums over Integer Ring
sage: FormalSums(ZZ[sqrt(2)])
Abelian Group of all Formal Finite Sums over Order in Number Field in sqrt2 with defining polyno
sage: FormalSums(GF(9,'a'))
Abelian Group of all Formal Finite Sums over Finite Field in a of size 3^2
```

base_extend(*R*)

EXAMPLES:

```
sage: FormalSums(ZZ).base_extend(GF(7))
Abelian Group of all Formal Finite Sums over Finite Field of size 7
```

CHAPTER

EIGHT

FACTORIZATIONS

The Factorization class provides a structure for holding quite general lists of objects with integer multiplicities. These may hold the results of an arithmetic or algebraic factorization, where the objects may be primes or irreducible polynomials and the multiplicities are the (non-zero) exponents in the factorization. For other types of examples, see below.

Factorization class objects contain a list, so can be printed nicely and be manipulated like a list of prime-exponent pairs, or easily turned into a plain list. For example, we factor the integer -45:

```
sage: F = factor(-45)
```

This returns an object of type Factorization:

```
sage: type(F)
<class 'sage.structure.factorization_integer.IntegerFactorization'>
```

It prints in a nice factored form:

```
sage: F
-1 * 3^2 * 5
```

There is an underlying list representation, emph{which ignores the unit part}:

```
sage: list(F)
[(3, 2), (5, 1)]
```

A Factorization is not actually a list:

```
sage: isinstance(F, list)
False
```

However, we can access the Factorization F itself as if it were a list:

```
sage: F[0]
(3, 2)
sage: F[1]
(5, 1)
```

To get at the unit part, use the Factorization.unit() function:

```
sage: F.unit()
-1
```

All factorizations are immutable, up to ordering with sort() and simplifying with simplify(). Thus if you write a function that returns a cached version of a factorization, you do not have to return a copy.

```
sage: F = factor(-12); F
-1 * 2^2 * 3
sage: F[0] = (5,4)
Traceback (most recent call last):
...
TypeError: 'Factorization' object does not support item assignment
```

EXAMPLES:

This more complicated example involving polynomials also illustrates that the unit part is not discarded from factorizations:

```
sage: x = QQ['x'].0
sage: f = -5*(x-2)*(x-3)
sage: f
-5*x^2 + 25*x - 30
sage: F = f.factor(); F
(-5) * (x - 3) * (x - 2)
sage: F.unit()
-5
sage: F.value()
-5*x^2 + 25*x - 30
```

The underlying list is the list of pairs (p_i, e_i), where each p_i is a 'prime' and each e_i is an integer. The unit part is discarded by the list:

```
sage: list(F)
[(x - 3, 1), (x - 2, 1)]
sage: len(F)
2
sage: F[1]
(x - 2, 1)
```

In the ring $\mathbf{Z}[x]$, the integer -5 is not a unit, so the factorization has three factors:

```
sage: x = ZZ['x'].0
sage: f = -5*(x-2)*(x-3)
sage: f
-5*x^2 + 25*x - 30
sage: F = f.factor(); F
(-1) * 5 * (x - 3) * (x - 2)
sage: F.universe()
Univariate Polynomial Ring in x over Integer Ring
sage: F.unit()
-1
sage: list(F)
[(5, 1), (x - 3, 1), (x - 2, 1)]
sage: F.value()
-5*x^2 + 25*x - 30
sage: len(F)
3
```

On the other hand, -1 is a unit in $\mathbf{Z}$, so it is included in the unit:

```
sage: x = ZZ['x'].0
sage: f = -1*(x-2)*(x-3)
sage: F = f.factor(); F
(-1) * (x - 3) * (x - 2)
```

```
sage: F.unit()
-1
sage: list(F)
[(x - 3, 1), (x - 2, 1)]
```

Factorizations can involve fairly abstract mathematical objects:

```
sage: F = ModularSymbols(11,4).factorization()
sage: F
(Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 6 for Gamma_0(11) of w
(Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 6 for Gamma_0(11) of w
(Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 6 for Gamma_0(11) of w
sage: type(F)
<class 'sage.structure.factorization.Factorization'>
```

```
sage: K.<a> = NumberField(x^2 + 3); K
Number Field in a with defining polynomial x^2 + 3
sage: f = K.factor(15); f
(Fractional ideal (-a))^2 * (Fractional ideal (5))
sage: f.universe()
Monoid of ideals of Number Field in a with defining polynomial x^2 + 3
sage: f.unit()
Fractional ideal (1)
sage: g=K.factor(9); g
(Fractional ideal (-a))^4
sage: f.lcm(g)
(Fractional ideal (-a))^4 * (Fractional ideal (5))
sage: f.gcd(g)
(Fractional ideal (-a))^2
sage: f.is_integral()
True
```

TESTS:

```
sage: F = factor(-20); F
-1 * 2^2 * 5
sage: G = loads(dumps(F)); G
-1 * 2^2 * 5
sage: G == F
True
sage: G is F
False
```

AUTHORS:

- William Stein (2006-01-22): added unit part as suggested by David Kohel.
- William Stein (2008-01-17): wrote much of the documentation and fixed a couple of bugs.
- Nick Alexander (2008-01-19): added support for non-commuting factors.
- John Cremona (2008-08-22): added division, lcm, gcd, is_integral and universe functions

class `sage.structure.factorization.`**`Factorization`**(*x*, *unit=None*, *cr=False*, *sort=True*, *simplify=True*)

Bases: `sage.structure.sage_object.SageObject`

A formal factorization of an object.

EXAMPLES:

```
sage: N = 2006
sage: F = N.factor(); F
2 * 17 * 59
sage: F.unit()
1
sage: F = factor(-2006); F
-1 * 2 * 17 * 59
sage: F.unit()
-1
sage: loads(F.dumps()) == F
True
sage: F = Factorization([(x,1/3)])
Traceback (most recent call last):
...
TypeError: exponents of factors must be integers
```

base_change(U)

Return the factorization self, with its factors (including the unit part) coerced into the universe U.

EXAMPLES:

```
sage: F = factor(2006)
sage: F.universe()
Integer Ring
sage: P.<x> = ZZ[]
sage: F.base_change(P).universe()
Univariate Polynomial Ring in x over Integer Ring
```

This method will return a TypeError if the coercion is not possible:

```
sage: g = x^2 - 1
sage: F = factor(g); F
(x - 1) * (x + 1)
sage: F.universe()
Univariate Polynomial Ring in x over Integer Ring
sage: F.base_change(ZZ)
Traceback (most recent call last):
...
TypeError: Impossible to coerce the factors of (x - 1) * (x + 1) into Integer Ring
```

expand()

Return the product of the factors in the factorization, multiplied out.

EXAMPLES:

```
sage: F = factor(-2006); F
-1 * 2 * 17 * 59
sage: F.value()
-2006

sage: R.<x,y> = FreeAlgebra(ZZ, 2)
sage: F = Factorization([(x,3), (y, 2), (x,1)]); F
x^3 * y^2 * x
sage: F.value()
x^3*y^2*x
```

gcd(*other*)

Return the gcd of two factorizations.

If the two factorizations have different universes, this method will attempt to find a common universe for the gcd. A TypeError is raised if this is impossible.

EXAMPLES:

```
sage: factor(-30).gcd(factor(-160))
2 * 5
sage: factor(gcd(-30,160))
2 * 5

sage: R.<x> = ZZ[]
sage: (factor(-20).gcd(factor(5*x+10))).universe()
Univariate Polynomial Ring in x over Integer Ring
```

is_commutative()

Return True if my factors commute.

EXAMPLES:

```
sage: F = factor(2006)
sage: F.is_commutative()
True
sage: K = QuadraticField(23, 'a')
sage: F = K.factor(13)
sage: F.is_commutative()
True
sage: R.<x,y,z> = FreeAlgebra(QQ, 3)
sage: F = Factorization([(z, 2)], 3)
sage: F.is_commutative()
False
sage: (F*F^-1).is_commutative()
False
```

is_integral()

Return True iff all exponents of this Factorization are non-negative.

EXAMPLES:

```
sage: F = factor(-10); F
-1 * 2 * 5
sage: F.is_integral()
True

sage: F = factor(-10) / factor(16); F
-1 * 2^-3 * 5
sage: F.is_integral()
False
```

lcm(*other*)

Return the lcm of two factorizations.

If the two factorizations have different universes, this method will attempt to find a common universe for the lcm. A TypeError is raised if this is impossible.

EXAMPLES:

```
sage: factor(-10).lcm(factor(-16))
2^4 * 5
sage: factor(lcm(-10,16))
2^4 * 5

sage: R.<x> = ZZ[]
sage: (factor(-20).lcm(factor(5*x+10))).universe()
```

```
Univariate Polynomial Ring in x over Integer Ring
```

prod()

Return the product of the factors in the factorization, multiplied out.

EXAMPLES:

```
sage: F = factor(-2006); F
-1 * 2 * 17 * 59
sage: F.value()
-2006

sage: R.<x,y> = FreeAlgebra(ZZ, 2)
sage: F = Factorization([(x,3), (y, 2), (x,1)]); F
x^3 * y^2 * x
sage: F.value()
x^3*y^2*x
```

radical()

Return the factorization of the radical of the value of self.

First, check that all exponents in the factorization are positive, raise ValueError otherwise. If all exponents are positive, return self with all exponents set to 1 and with the unit set to 1.

EXAMPLES:

```
sage: F = factor(-100); F
-1 * 2^2 * 5^2
sage: F.radical()
2 * 5
sage: factor(1/2).radical()
Traceback (most recent call last):
...
ValueError: All exponents in the factorization must be positive.
```

radical_value()

Return the product of the prime factors in self.

First, check that all exponents in the factorization are positive, raise ValueError otherwise. If all exponents are positive, return the product of the prime factors in self. This should be functionally equivalent to self.radical().value()

EXAMPLES:

```
sage: F = factor(-100); F
-1 * 2^2 * 5^2
sage: F.radical_value()
10
sage: factor(1/2).radical_value()
Traceback (most recent call last):
...
ValueError: All exponents in the factorization must be positive.
```

simplify()

Combine adjacent products as much as possible.

TESTS:

```
sage: R.<x,y> = FreeAlgebra(ZZ, 2)
sage: F = Factorization([(x,3), (y, 2), (y,2)], simplify=False); F
x^3 * y^2 * y^2
```

```
sage: F.simplify(); F
x^3 * y^4
sage: F * Factorization([(y, -2)], 2)
(2) * x^3 * y^2
```

sort (*_cmp=None*)

Sort the factors in this factorization.

INPUT:

- `_cmp` - (default: None) comparison function

OUTPUT:

- changes this factorization to be sorted

If _cmp is None, we determine the comparison function as follows: If the prime in the first factor has a dimension method, then we sort based first on *dimension* then on the exponent. If there is no dimension method, we next attempt to sort based on a degree method, in which case, we sort based first on *degree*, then exponent to break ties when two factors have the same degree, and if those match break ties based on the actual prime itself. If there is no degree method, we sort based on dimension.

EXAMPLES:

We create a factored polynomial:

```
sage: x = polygen(QQ,'x')
sage: F = factor(x^3 + 1); F
(x + 1) * (x^2 - x + 1)
```

Then we sort it but using the negated version of the standard Python cmp function:

```
sage: F.sort(_cmp = lambda x,y: -cmp(x,y))
sage: F
(x^2 - x + 1) * (x + 1)
```

unit ()

Return the unit part of this factorization.

EXAMPLES:

We create a polynomial over the real double field and factor it:

```
sage: x = polygen(RDF, 'x')
sage: F = factor(-2*x^2 - 1); F
(-2.0) * (x^2 + 0.5000000000000001)
```

Note that the unit part of the factorization is -2.0:

```
sage: F.unit()
-2.0

sage: F = factor(-2006); F
-1 * 2 * 17 * 59
sage: F.unit()
-1
```

universe ()

Return the parent structure of my factors.

Note: This used to be called `base_ring`, but the universe of a factorization need not be a ring.

EXAMPLES:

```
sage: F = factor(2006)
sage: F.universe()
Integer Ring

sage: R.<x,y,z> = FreeAlgebra(QQ, 3)
sage: F = Factorization([(z, 2)], 3)
sage: (F*F^-1).universe()
Free Algebra on 3 generators (x, y, z) over Rational Field

sage: F = ModularSymbols(11,4).factorization()
sage: F.universe()
```

value()

Return the product of the factors in the factorization, multiplied out.

EXAMPLES:

```
sage: F = factor(-2006); F
-1 * 2 * 17 * 59
sage: F.value()
-2006

sage: R.<x,y> = FreeAlgebra(ZZ, 2)
sage: F = Factorization([(x,3), (y, 2), (x,1)]); F
x^3 * y^2 * x
sage: F.value()
x^3*y^2*x
```

CHAPTER

NINE

ELEMENTS

AUTHORS:

- David Harvey (2006-10-16): changed CommutativeAlgebraElement to derive from CommutativeRingElement instead of AlgebraElement
- David Harvey (2006-10-29): implementation and documentation of new arithmetic architecture
- William Stein (2006-11): arithmetic architecture – pushing it through to completion.
- Gonzalo Tornaria (2007-06): recursive base extend for coercion – lots of tests
- Robert Bradshaw (2007-2010): arithmetic operators and coercion
- Maarten Derickx (2010-07): added architecture for is_square and sqrt

9.1 The Abstract Element Class Hierarchy

This is the abstract class hierarchy, i.e., these are all abstract base classes.

```
SageObject
    Element
        ModuleElement
            RingElement
                CommutativeRingElement
                    IntegralDomainElement
                        DedekindDomainElement
                            PrincipalIdealDomainElement
                                EuclideanDomainElement
                    FieldElement
                        FiniteFieldElement
                    CommutativeAlgebraElement
                AlgebraElement   (note -- can't derive from module, since no multiple inheritance)
                    CommutativeAlgebra ??? (should be removed from element.pxd)
                    Matrix
                InfinityElement
                    PlusInfinityElement
                    MinusInfinityElement
            AdditiveGroupElement
            Vector

        MonoidElement
            MultiplicativeGroupElement
    ElementWithCachedMethod
```

9.2 How to Define a New Element Class

Elements typically define a method _new_c, e.g.,

```
cdef _new_c(self, defining data):
    cdef FreeModuleElement_generic_dense x
    x = FreeModuleElement_generic_dense.__new__(FreeModuleElement_generic_dense)
    x._parent = self._parent
    x._entries = v
```

that creates a new sibling very quickly from defining data with assumed properties.

Sage has a special system in place for handling arithmetic operations for all Element subclasses. There are various rules that must be followed by both arithmetic implementers and callers.

A quick summary for the impatient:

- To implement addition for any Element class, override `def _add_()`.
- If you want to add `x` and `y`, whose parents you know are **identical**, you may call `_add_(x, y)`. This will be the fastest way to guarantee that the correct implementation gets called. Of course you can still always use `x + y`.

Now in more detail. The aims of this system are to provide (1) an efficient calling protocol from both Python and Cython, (2) uniform coercion semantics across Sage, (3) ease of use, (4) readability of code.

We will take addition of RingElements as an example; all other operators and classes are similar. There are three relevant functions, with subtly differing names (`add` vs. `iadd`, single vs. double underscores).

- **def RingElement.__add__**

  This function is called by Python or Cython when the binary "+" operator is encountered. It **assumes** that at least one of its arguments is a RingElement; only a really twisted programmer would violate this condition. It has a fast pathway to deal with the most common case where the arguments have the same parent. Otherwise, it uses the coercion module to work out how to make them have the same parent. After any necessary coercions have been performed, it calls `_add_` to dispatch to the correct underlying addition implementation.

  Note that although this function is declared as `def`, it doesn't have the usual overheads associated with Python functions (either for the caller or for `__add__` itself). This is because Python has optimised calling protocols for such special functions.

- **def RingElement._add_**

  This is the function you should override to implement addition in a subclass of `RingElement`.

  The two arguments to this function are guaranteed to have the **same parent**. Its return value **must** have the **same parent** as its arguments.

  If you want to add two objects and you know that their parents are the same object, you are encouraged to call this function directly, instead of using `x + y`.

  When implementing `_add_` in a Cython extension class, use `cpdef _add_` instead of `def _add_`.

For speed, there are also *inplace* versions of the arithmetic commands. **Do not** call them directly, they may mutate the object and will be called when and only when it has been determined that the old object will no longer be accessible from the calling function after this operation.

- **def RingElement._iadd_**

  This is the function you should override to implement inplace addition in a Python subclass of RingElement.

  The two arguments to this function are guaranteed to have the **same parent**. Its return value **must** have the **same parent** as its arguments.

The default implementation of this function is to call _add_, so if no one has defined a Python implementation, the correct Cython implementation will get called.

class sage.structure.element.**AdditiveGroupElement**
Bases: sage.structure.element.ModuleElement

Generic element of an additive group.

order()
Return additive order of element

class sage.structure.element.**AlgebraElement**
Bases: sage.structure.element.RingElement

INPUT:

- parent - a SageObject

class sage.structure.element.**CoercionModel**
Bases: object

Most basic coercion scheme. If it doesn't already match, throw an error.

bin_op(*x*, *y*, *op*)

canonical_coercion(*x*, *y*)

class sage.structure.element.**CommutativeAlgebra**
Bases: sage.structure.element.AlgebraElement

INPUT:

- parent - a SageObject

class sage.structure.element.**CommutativeAlgebraElement**
Bases: sage.structure.element.CommutativeRingElement

INPUT:

- parent - a SageObject

class sage.structure.element.**CommutativeRingElement**
Bases: sage.structure.element.RingElement

Base class for elements of commutative rings.

divides(*x*)
Return True if self divides x.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(QQ)
sage: x.divides(x^2)
True
sage: x.divides(x^2+2)
False
sage: (x^2+2).divides(x)
False
sage: P.<x> = PolynomialRing(ZZ)
sage: x.divides(x^2)
True
sage: x.divides(x^2+2)
False
sage: (x^2+2).divides(x)
False
```

trac ticket #5347 has been fixed:

```
sage: K = GF(7)
sage: K(3).divides(1)
True
sage: K(3).divides(K(1))
True

sage: R = Integers(128)
sage: R(0).divides(1)
False
sage: R(0).divides(0)
True
sage: R(0).divides(R(0))
True
sage: R(1).divides(0)
True
sage: R(121).divides(R(120))
True
sage: R(120).divides(R(121))
Traceback (most recent call last):
...
ZeroDivisionError: reduction modulo right not defined.
```

If x has different parent than self, they are first coerced to a common parent if possible. If this coercion fails, it returns a TypeError. This fixes trac ticket #5759.

```
sage: Zmod(2)(0).divides(Zmod(2)(0))
True
sage: Zmod(2)(0).divides(Zmod(2)(1))
False
sage: Zmod(5)(1).divides(Zmod(2)(1))
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents: 'Ring of integers modulo 5'
sage: Zmod(35)(4).divides(Zmod(7)(1))
True
sage: Zmod(35)(7).divides(Zmod(7)(1))
False
```

inverse_mod(*I*)

Return an inverse of self modulo the ideal I, if defined, i.e., if I and self together generate the unit ideal.

is_square(*root=False*)

Return whether or not the ring element self is a square.

If the optional argument root is True, then also return the square root (or None, if it is not a square).

INPUT:

- root - whether or not to also return a square root (default: False)

OUTPUT:

- bool – whether or not a square
- object – (optional) an actual square root if found, and None otherwise.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = 12*(x+1)^2 * (x+3)^2
```

```
sage: f.is_square()
False
sage: f.is_square(root=True)
(False, None)
sage: h = f/3
sage: h.is_square()
True
sage: h.is_square(root=True)
(True, 2*x^2 + 8*x + 6)
```

Note: This is the is_square implementation for general commutative ring elements. It's implementation is to raise a NotImplementedError. The function definition is here to show what functionality is expected and provide a general framework.

mod(*I*)

Return a representative for `self` modulo the ideal I (or the ideal generated by the elements of I if I is not an ideal.)

EXAMPLE: Integers Reduction of 5 modulo an ideal:

```
sage: n = 5
sage: n.mod(3*ZZ)
2
```

Reduction of 5 modulo the ideal generated by 3:

```
sage: n.mod(3)
2
```

Reduction of 5 modulo the ideal generated by 15 and 6, which is (3).

```
sage: n.mod([15,6])
2
```

EXAMPLE: Univariate polynomials

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = x^3 + x + 1
sage: f.mod(x + 1)
-1
```

Reduction for $\mathbf{Z}[x]$:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = x^3 + x + 1
sage: f.mod(x + 1)
-1
```

When little is implemented about a given ring, then mod may return simply return f.

EXAMPLE: Multivariate polynomials We reduce a polynomial in two variables modulo a polynomial and an ideal:

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: (x^2 + y^2 + z^2).mod(x+y+z)
2*y^2 + 2*y*z + 2*z^2
```

Notice above that x is eliminated. In the next example, both y and z are eliminated:

```
sage: (x^2 + y^2 + z^2).mod( (x - y, y - z) )
3*z^2
sage: f = (x^2 + y^2 + z^2)^2; f
x^4 + 2*x^2*y^2 + y^4 + 2*x^2*z^2 + 2*y^2*z^2 + z^4
sage: f.mod( (x - y, y - z) )
9*z^4
```

In this example y is eliminated:

```
sage: (x^2 + y^2 + z^2).mod( (x^3, y - z) )
x^2 + 2*z^2
```

sqrt (*extend=True*, *all=False*, *name=None*)

It computes the square root.

INPUT:

- `extend` - Whether to make a ring extension containing a square root if `self` is not a square (default: `True`)
- `all` - Whether to return a list of all square roots or just a square root (default: False)
- `name` - Required when `extend=True` and `self` is not a square. This will be the name of the generator extension.

OUTPUT:

- if `all=False` it returns a square root. (throws an error if `extend=False` and `self` is not a square)
- if `all=True` it returns a list of all the square roots (could be empty if `extend=False` and `self` is not a square)

ALGORITHM:

It uses `is_square(root=true)` for the hard part of the work, the rest is just wrapper code.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: (x^2).sqrt()
x
sage: f=x^2-4*x+4; f.sqrt(all=True)
[x - 2, -x + 2]
sage: sqrtx=x.sqrt(name="y"); sqrtx
y
sage: sqrtx^2
x
sage: x.sqrt(all=true,name="y")
[y, -y]
sage: x.sqrt(extend=False,all=True)
[]
sage: x.sqrt()
Traceback (most recent call last):
...
TypeError: Polynomial is not a square. You must specify the name of the square root when usi
sage: x.sqrt(extend=False)
Traceback (most recent call last):
...
ValueError: trying to take square root of non-square x with extend = False
```

TESTS:

```
sage: f = (x+3)^2; f.sqrt()
x + 3
sage: f = (x+3)^2; f.sqrt(all=True)
[x + 3, -x - 3]
sage: f = (x^2 - x + 3)^2; f.sqrt()
x^2 - x + 3
sage: f = (x^2 - x + 3)^6; f.sqrt()
x^6 - 3*x^5 + 12*x^4 - 19*x^3 + 36*x^2 - 27*x + 27
sage: g = (R.random_element(15))^2
sage: g.sqrt()^2 == g
True

sage: R.<x> = GF(250037)[]
sage: f = x^2/(x+1)^2; f.sqrt()
x/(x + 1)
sage: f = 9 * x^4 / (x+1)^2; f.sqrt()
3*x^2/(x + 1)
sage: f = 9 * x^4 / (x+1)^2; f.sqrt(all=True)
[3*x^2/(x + 1), 250034*x^2/(x + 1)]

sage: R.<x> = QQ[]
sage: a = 2*(x+1)^2 / (2*(x-1)^2); a.sqrt()
(2*x + 2)/(2*x - 2)
sage: sqrtx=(1/x).sqrt(name="y"); sqrtx
y
sage: sqrtx^2
1/x
sage: (1/x).sqrt(all=true,name="y")
[y, -y]
sage: (1/x).sqrt(extend=False,all=True)
[]
sage: (1/(x^2-1)).sqrt()
Traceback (most recent call last):
...
TypeError: Polynomial is not a square. You must specify the name of the square root when usi
sage: (1/(x^2-3)).sqrt(extend=False)
Traceback (most recent call last):
...
ValueError: trying to take square root of non-square 1/(x^2 - 3) with extend = False
```

class sage.structure.element.**DedekindDomainElement**

Bases: sage.structure.element.IntegralDomainElement

INPUT:

- parent - a SageObject

class sage.structure.element.**Element**

Bases: sage.structure.sage_object.SageObject

Generic element of a structure. All other types of elements (RingElement, ModuleElement, etc) derive from this type.

Subtypes must either call `__init__()` to set `_parent`, or may set `_parent` themselves if that would be more efficient.

N(*prec=None*, *digits=None*, *algorithm=None*)

Return a numerical approximation of x with at least prec bits of precision.

EXAMPLES:

```
sage: (2/3).n()
0.666666666666667
sage: pi.n(digits=10)   # indirect doctest
3.141592654
sage: pi.n(prec=20)    # indirect doctest
3.1416
```

TESTS:

Check that trac ticket #14778 is fixed:

```
sage: (0).n(algorithm='foo')
0.000000000000000
```

base_extend(*R*)

base_ring()

Return the base ring of this element's parent (if that makes sense).

TESTS:

```
sage: QQ.base_ring()
Rational Field
sage: identity_matrix(3).base_ring()
Integer Ring
```

category()

is_zero()

Return `True` if `self` equals `self.parent()(0)`.

The default implementation is to fall back to `not self.__nonzero__`.

> **Warning:** Do not re-implement this method in your subclass but implement `__nonzero__` instead.

n(*prec=None*, *digits=None*, *algorithm=None*)

Return a numerical approximation of x with at least prec bits of precision.

EXAMPLES:

```
sage: (2/3).n()
0.666666666666667
sage: pi.n(digits=10)   # indirect doctest
3.141592654
sage: pi.n(prec=20)    # indirect doctest
3.1416
```

TESTS:

Check that trac ticket #14778 is fixed:

```
sage: (0).n(algorithm='foo')
0.000000000000000
```

numerical_approx(*prec=None*, *digits=None*, *algorithm=None*)

Return a numerical approximation of x with at least prec bits of precision.

EXAMPLES:

```
sage: (2/3).n()
0.666666666666667
sage: pi.n(digits=10)   # indirect doctest
```

```
3.141592654
sage: pi.n(prec=20)    # indirect doctest
3.1416
```

TESTS:

Check that trac ticket #14778 is fixed:

```
sage: (0).n(algorithm='foo')
0.000000000000000
```

parent (*x=None*)
Return the parent of this element; or, if the optional argument x is supplied, the result of coercing x into the parent of this element.

subs (*in_dict=None*, ***kwds*)
Substitutes given generators with given values while not touching other generators. This is a generic wrapper around `__call__`. The syntax is meant to be compatible with the corresponding method for symbolic expressions.

INPUT:

- `in_dict` - (optional) dictionary of inputs
- `**kwds` - named parameters

OUTPUT:

- new object if substitution is possible, otherwise self.

EXAMPLES:

```
sage: x, y = PolynomialRing(ZZ,2,'xy').gens()
sage: f = x^2 + y + x^2*y^2 + 5
sage: f((5,y))
25*y^2 + y + 30
sage: f.subs({x:5})
25*y^2 + y + 30
sage: f.subs(x=5)
25*y^2 + y + 30
sage: (1/f).subs(x=5)
1/(25*y^2 + y + 30)
sage: Integer(5).subs(x=4)
5
```

substitute (*in_dict=None*, ***kwds*)
This is an alias for self.subs().

INPUT:

- `in_dict` - (optional) dictionary of inputs
- `**kwds` - named parameters

OUTPUT:

- new object if substitution is possible, otherwise self.

EXAMPLES:

```
sage: x, y = PolynomialRing(ZZ,2,'xy').gens()
sage: f = x^2 + y + x^2*y^2 + 5
sage: f((5,y))
25*y^2 + y + 30
```

```
sage: f.substitute({x:5})
25*y^2 + y + 30
sage: f.substitute(x=5)
25*y^2 + y + 30
sage: (1/f).substitute(x=5)
1/(25*y^2 + y + 30)
sage: Integer(5).substitute(x=4)
5
```

class sage.structure.element.**ElementWithCachedMethod**

Bases: sage.structure.element.Element

An element class that fully supports cached methods.

NOTE:

The cached_method decorator provides a convenient way to automatically cache the result of a computation. Since trac ticket #11115, the cached method decorator applied to a method without optional arguments is faster than a hand-written cache in Python, and a cached method without any arguments (except self) is actually faster than a Python method that does nothing more but to return 1. A cached method can also be inherited from the parent or element class of a category.

However, this holds true only if attribute assignment is supported. If you write an extension class in Cython that does not accept attribute assignment then a cached method inherited from the category will be slower (for Parent) or the cache would even break (for Element).

This class should be used if you write an element class, can not provide it with attribute assignment, but want that it inherits a cached method from the category. Under these conditions, your class should inherit from this class rather than Element. Then, the cache will work, but certainly slower than with attribute assignment. Lazy attributes work as well.

EXAMPLE:

We define three element extension classes. The first inherits from Element, the second from this class, and the third simply is a Python class. We also define a parent class and, in Python, a category whose element and parent classes define cached methods.

```
sage: cython_code = ["from sage.structure.element cimport Element, ElementWithCachedMethod",
... "cdef class MyBrokenElement(Element):",
... "    cdef public object x",
... "    def __init__(self,P,x):",
... "        self.x=x",
... "        Element.__init__(self,P)",
... "    def __neg__(self):",
... "        return MyBrokenElement(self.parent(),-self.x)",
... "    def _repr_(self):",
... "        return '<%s>'%self.x",
... "    def __hash__(self):",
... "        return hash(self.x)",
... "    def __cmp__(left, right):",
... "        return (<Element>left)._cmp(right)",
... "    def __richcmp__(left, right, op):",
... "        return (<Element>left)._richcmp(right,op)",
... "    cdef int _cmp_c_impl(left, Element right) except -2:",
... "        return cmp(left.x,right.x)",
... "    def raw_test(self):",
... "        return -self",
... "cdef class MyElement(ElementWithCachedMethod):",
... "    cdef public object x",
... "    def __init__(self,P,x):",
```

```
... "        self.x=x",
... "        Element.__init__(self,P)",
... "    def __neg__(self):",
... "        return MyElement(self.parent(),-self.x)",
... "    def _repr_(self):",
... "        return '<%s>'%self.x",
... "    def __hash__(self):",
... "        return hash(self.x)",
... "    def __cmp__(left, right):",
... "        return (<Element>left)._cmp(right)",
... "    def __richcmp__(left, right, op):",
... "        return (<Element>left)._richcmp(right,op)",
... "    cdef int _cmp_c_impl(left, Element right) except -2:",
... "        return cmp(left.x,right.x)",
... "    def raw_test(self):",
... "        return -self",
... "class MyPythonElement(MyBrokenElement): pass",
... "from sage.structure.parent cimport Parent",
... "cdef class MyParent(Parent):",
... "    Element = MyElement"]
sage: cython('\n'.join(cython_code))
sage: cython_code = ["from sage.all import cached_method, cached_in_parent_method, Category, Obj
... "class MyCategory(Category):",
... "    @cached_method",
... "    def super_categories(self):",
... "        return [Objects()]",
... "    class ElementMethods:",
... "        @cached_method",
... "        def element_cache_test(self):",
... "            return -self",
... "        @cached_in_parent_method",
... "        def element_via_parent_test(self):",
... "            return -self",
... "    class ParentMethods:",
... "        @cached_method",
... "        def one(self):",
... "            return self.element_class(self,1)",
... "        @cached_method",
... "        def invert(self, x):",
... "            return -x"]
sage: cython('\n'.join(cython_code))
sage: C = MyCategory()
sage: P = MyParent(category=C)
sage: ebroken = MyBrokenElement(P,5)
sage: e = MyElement(P,5)
```

The cached methods inherited by `MyElement` works:

```
sage: e.element_cache_test()
<-5>
sage: e.element_cache_test() is e.element_cache_test()
True
sage: e.element_via_parent_test()
<-5>
sage: e.element_via_parent_test() is e.element_via_parent_test()
True
```

The other element class can only inherit a `cached_in_parent_method`, since the cache is stored in the parent. In fact, equal elements share the cache, even if they are of different types:

```
sage: e == ebroken
True
sage: type(e) == type(ebroken)
False
sage: ebroken.element_via_parent_test() is e.element_via_parent_test()
True
```

However, the cache of the other inherited method breaks, although the method as such works:

```
sage: ebroken.element_cache_test()
<-5>
sage: ebroken.element_cache_test() is ebroken.element_cache_test()
False
```

Since `e` and `ebroken` share the cache, when we empty it for one element it is empty for the other as well:

```
sage: b = ebroken.element_via_parent_test()
sage: e.element_via_parent_test.clear_cache()
sage: b is ebroken.element_via_parent_test()
False
```

Note that the cache only breaks for elements that do no allow attribute assignment. A Python version of `MyBrokenElement` therefore allows for cached methods:

```
sage: epython = MyPythonElement(P,5)
sage: epython.element_cache_test()
<-5>
sage: epython.element_cache_test() is epython.element_cache_test()
True
```

class `sage.structure.element.`**`EuclideanDomainElement`**

Bases: `sage.structure.element.PrincipalIdealDomainElement`

INPUT:

- `parent` - a SageObject

`degree`()

`leading_coefficient`()

`quo_rem`(*other*)

class `sage.structure.element.`**`FieldElement`**

Bases: `sage.structure.element.CommutativeRingElement`

INPUT:

- `parent` - a SageObject

`divides`(*other*)

Check whether `self` divides other, for field elements.

Since this is a field, all values divide all other values, except that zero does not divide any non-zero values.

EXAMPLES:

```
sage: K.<rt3> = QQ[sqrt(3)]
sage: K(0).divides(rt3)
False
sage: rt3.divides(K(17))
True
sage: K(0).divides(K(0))
```

```
True
sage: rt3.divides(K(0))
True
```

is_unit()

Return `True` if `self` is a unit in its parent ring.

EXAMPLES:

```
sage: a = 2/3; a.is_unit()
True
```

On the other hand, 2 is not a unit, since its parent is $\mathbf{Z}$.

```
sage: a = 2; a.is_unit()
False
sage: parent(a)
Integer Ring
```

However, a is a unit when viewed as an element of QQ:

```
sage: a = QQ(2); a.is_unit()
True
```

quo_rem(*right*)

Return the quotient and remainder obtained by dividing `self` by `right`. Since this element lives in a field, the remainder is always zero and the quotient is `self/right`.

TESTS:

Test if trac ticket #8671 is fixed:

```
sage: R.<x,y> = QQ[]
sage: S.<a,b> = R.quo(y^2 + 1)
sage: S.is_field = lambda : False
sage: F = Frac(S); u = F.one()
sage: u.quo_rem(u)
(1, 0)
```

class `sage.structure.element.`**InfinityElement**

Bases: `sage.structure.element.RingElement`

INPUT:

- `parent` - a SageObject

class `sage.structure.element.`**IntegralDomainElement**

Bases: `sage.structure.element.CommutativeRingElement`

INPUT:

- `parent` - a SageObject

is_nilpotent()

class `sage.structure.element.`**Matrix**

Bases: `sage.structure.element.ModuleElement`

INPUT:

- `parent` - a SageObject

class `sage.structure.element.`**MinusInfinityElement**

Bases: `sage.structure.element.InfinityElement`

INPUT:

- parent - a SageObject

class sage.structure.element.**ModuleElement**

Bases: sage.structure.element.Element

Generic element of a module.

additive_order()

Return the additive order of self.

order()

Return the additive order of self.

class sage.structure.element.**MonoidElement**

Bases: sage.structure.element.Element

Generic element of a monoid.

multiplicative_order()

Return the multiplicative order of self.

order()

Return the multiplicative order of self.

powers(n)

Return the list $[x^0, x^1, \ldots, x^{n-1}]$.

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: g = G([2, 3, 4, 1])
sage: g.powers(4)
[(), (1,2,3,4), (1,3)(2,4), (1,4,3,2)]
```

class sage.structure.element.**MultiplicativeGroupElement**

Bases: sage.structure.element.MonoidElement

Generic element of a multiplicative group.

order()

Return the multiplicative order of self.

class sage.structure.element.**NamedBinopMethod**

Bases: object

A decorator to be used on binary operation methods that should operate on elements of the same parent. If the parents of the arguments differ, coercion is performed, then the method is re-looked up by name on the first argument.

In short, using the NamedBinopMethod (alias coerce_binop) decorator on a method gives it the exact same semantics of the basic arithmetic operations like _add_, _sub_, etc. in that both operands are guaranteed to have exactly the same parent.

class sage.structure.element.**PlusInfinityElement**

Bases: sage.structure.element.InfinityElement

INPUT:

- parent - a SageObject

class sage.structure.element.**PrincipalIdealDomainElement**

Bases: sage.structure.element.DedekindDomainElement

INPUT:

- `parent` - a SageObject

lcm(*right*)

Return the least common multiple of `self` and right.

class sage.structure.element.**RingElement**

Bases: `sage.structure.element.ModuleElement`

INPUT:

- `parent` - a SageObject

abs()

Return the absolute value of `self`. (This just calls the `__abs__` method, so it is equivalent to the `abs()` built-in function.)

EXAMPLES:

```
sage: RR(-1).abs()
1.00000000000000
sage: ZZ(-1).abs()
1
sage: CC(I).abs()
1.00000000000000
sage: Mod(-15, 37).abs()
Traceback (most recent call last):
...
ArithmeticError: absolute valued not defined on integers modulo n.
```

additive_order()

Return the additive order of `self`.

is_nilpotent()

Return `True` if `self` is nilpotent, i.e., some power of `self` is 0.

TESTS:

```
sage: a = QQ(2)
sage: a.is_nilpotent()
False
sage: a = QQ(0)
sage: a.is_nilpotent()
True
sage: m = matrix(QQ,3,[[3,2,3],[9,0,3],[-9,0,-3]])
sage: m.is_nilpotent()
Traceback (most recent call last):
...
AttributeError: ... object has no attribute 'is_nilpotent'
```

is_one()

is_prime()

Is `self` a prime element?

A *prime* element is a non-zero, non-unit element p such that, whenever p divides ab for some a and b, then p divides a or p divides b.

EXAMPLES:

For polynomial rings, prime is the same as irreducible:

```
sage: R.<x,y> = QQ[]
sage: x.is_prime()
True
sage: (x^2 + y^3).is_prime()
True
sage: (x^2 - y^2).is_prime()
False
sage: R(0).is_prime()
False
sage: R(2).is_prime()
False
```

For the Gaussian integers:

```
sage: K.<i> = QuadraticField(-1)
sage: ZI = K.ring_of_integers()
sage: ZI(3).is_prime()
True
sage: ZI(5).is_prime()
False
sage: ZI(2+i).is_prime()
True
sage: ZI(0).is_prime()
False
sage: ZI(1).is_prime()
False
```

In fields, an element is never prime:

```
sage: RR(0).is_prime()
False
sage: RR(2).is_prime()
False
```

For integers, prime numbers are redefined to be positive:

```
sage: RingElement.is_prime(-2)
True
sage: Integer.is_prime(-2)
False
```

multiplicative_order()

Return the multiplicative order of `self`, if `self` is a unit, or raise `ArithmeticError` otherwise.

powers(*n*)

Return the list $[x^0, x^1, \ldots, x^{n-1}]$.

EXAMPLES:

```
sage: 5.powers(3)
[1, 5, 25]
```

class sage.structure.element.**Vector**

Bases: `sage.structure.element.ModuleElement`

INPUT:

- `parent` - a SageObject

sage.structure.element.**bin_op**(*x*, *y*, *op*) sage.structure.element.**canonical_coercion**(*x*, *y*)

canonical_coercion(x,y) is what is called before doing an arithmetic operation between x and y. It returns a pair (z,w) such that z is got from x and w from y via canonical coercion and the parents of z and w are identical.

EXAMPLES:

```
sage: A = Matrix([[0, 1], [1, 0]])
sage: canonical_coercion(A, 1)
(
[0 1]  [1 0]
[1 0], [0 1]
)
```

sage.structure.element.**coerce_binop**

alias of NamedBinopMethod

sage.structure.element.**coerce_cmp**(*x*, *y*)

sage.structure.element.**coercion_traceback**(*dump=True*)

This function is very helpful in debugging coercion errors. It prints the tracebacks of all the errors caught in the coercion detection. Note that failure is cached, so some errors may be omitted the second time around (as it remembers not to retry failed paths for speed reasons.

For performance and caching reasons, exception recording must be explicitly enabled before using this function.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.record_exceptions()
sage: 1 + 1/5
6/5
sage: coercion_traceback()  # Should be empty, as all went well.
sage: 1/5 + GF(5).gen()
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '+': 'Rational Field' and 'Finite Field of size 5'
sage: coercion_traceback()
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents: 'Rational Field' and 'Finite Fie
```

sage.structure.element.**generic_power**(*a*, *n*, *one=None*)

Computes a^n, where n is an integer, and a is an object which supports multiplication. Optionally an additional argument, which is used in the case that n == 0:

- one - the "unit" element, returned directly (can be anything)

If this is not supplied, int(1) is returned.

EXAMPLES:

```
sage: from sage.structure.element import generic_power
sage: generic_power(int(12),int(0))
1
sage: generic_power(int(0),int(100))
0
sage: generic_power(Integer(10),Integer(0))
1
sage: generic_power(Integer(0),Integer(23))
0
sage: sum([generic_power(2,i) for i in range(17)]) #test all 4-bit combinations
```

```
131071
sage: F = Zmod(5)
sage: a = generic_power(F(2), 5); a
2
sage: a.parent() is F
True
sage: a = generic_power(F(1), 2)
sage: a.parent() is F
True

sage: generic_power(int(5), 0)
1
```

sage.structure.element.**get_coercion_model**()

Return the global coercion model.

EXAMPLES:

```
sage: import sage.structure.element as e
sage: cm = e.get_coercion_model()
sage: cm
<sage.structure.coerce.CoercionModel_cache_maps object at ...>
```

sage.structure.element.**have_same_parent**(*left*, *right*)

Return `True` if and only if `left` and `right` have the same parent.

> **Warning:** This function assumes that at least one of the arguments is a Sage `Element`. When in doubt, use the slower `parent(left) is parent(right)` instead.

EXAMPLES:

```
sage: from sage.structure.element import have_same_parent
sage: have_same_parent(1, 3)
True
sage: have_same_parent(1, 1/2)
False
sage: have_same_parent(gap(1), gap(1/2))
True
```

These have different types but the same parent:

```
sage: a = RLF(2)
sage: b = exp(a)
sage: type(a)
<type 'sage.rings.real_lazy.LazyWrapper'>
sage: type(b)
<type 'sage.rings.real_lazy.LazyNamedUnop'>
sage: have_same_parent(a, b)
True
```

sage.structure.element.**is_AdditiveGroupElement**(*x*)

Return `True` if x is of type AdditiveGroupElement.

sage.structure.element.**is_AlgebraElement**(*x*)

Return `True` if x is of type AlgebraElement.

TESTS:

```
sage: from sage.structure.element import is_AlgebraElement
sage: R.<x,y> = FreeAlgebra(QQ,2)
```

```
sage: is_AlgebraElement(x*y)
True

sage: is_AlgebraElement(1)
False
```

sage.structure.element.**is_CommutativeAlgebraElement**(*x*)
Return `True` if x is of type CommutativeAlgebraElement.

sage.structure.element.**is_CommutativeRingElement**(*x*)
Return `True` if x is of type CommutativeRingElement.

TESTS:

```
sage: from sage.rings.commutative_ring_element import is_CommutativeRingElement
sage: is_CommutativeRingElement(oo)
False

sage: is_CommutativeRingElement(1)
True
```

sage.structure.element.**is_DedekindDomainElement**(*x*)
Return `True` if x is of type DedekindDomainElement.

sage.structure.element.**is_Element**(*x*)
Return `True` if x is of type Element.

EXAMPLES:

```
sage: from sage.structure.element import is_Element
sage: is_Element(2/3)
True
sage: is_Element(QQ^3)
False
```

sage.structure.element.**is_EuclideanDomainElement**(*x*)
Return `True` if x is of type EuclideanDomainElement.

sage.structure.element.**is_FieldElement**(*x*)
Return `True` if x is of type FieldElement.

sage.structure.element.**is_InfinityElement**(*x*)
Return `True` if x is of type InfinityElement.

TESTS:

```
sage: from sage.structure.element import is_InfinityElement
sage: is_InfinityElement(1)
False

sage: is_InfinityElement(oo)
True
```

sage.structure.element.**is_IntegralDomainElement**(*x*)
Return `True` if x is of type IntegralDomainElement.

sage.structure.element.**is_Matrix**(*x*)

sage.structure.element.**is_ModuleElement**(*x*)
Return `True` if x is of type ModuleElement.

This is even faster than using isinstance inline.

EXAMPLES:

```
sage: from sage.structure.element import is_ModuleElement
sage: is_ModuleElement(2/3)
True
sage: is_ModuleElement((QQ^3).0)
True
sage: is_ModuleElement('a')
False
```

sage.structure.element.**is_MonoidElement**(*x*)
Return `True` if x is of type MonoidElement.

sage.structure.element.**is_MultiplicativeGroupElement**(*x*)
Return `True` if x is of type MultiplicativeGroupElement.

sage.structure.element.**is_PrincipalIdealDomainElement**(*x*)
Return `True` if x is of type PrincipalIdealDomainElement.

sage.structure.element.**is_RingElement**(*x*)
Return `True` if x is of type RingElement.

sage.structure.element.**is_Vector**(*x*)

sage.structure.element.**make_element**(*_class*, *_dict*, *parent*)
This function is only here to support old pickles.

Pickling functionality is moved to Element.{__getstate__,__setstate__} functions.

sage.structure.element.**parent**(*x*)
Return the parent of the element `x`.

Usually, this means the mathematical object of which `x` is an element.

INPUT:

- `x` – an element

OUTPUT:

- if `x` is a Sage `Element`, return `x.parent()`.
- if `x` has a `parent` method and `x` does not have an `__int__` or `__float__` method, return `x.parent()`.
- otherwise, return `type(x)`.

See also:

Parents, Conversion and Coercion Section in the Sage Tutorial

EXAMPLES:

```
sage: a = 42
sage: parent(a)
Integer Ring
sage: b = 42/1
sage: parent(b)
Rational Field
sage: c = 42.0
sage: parent(c)
Real Field with 53 bits of precision
```

Some more complicated examples:

```
sage: x = Partition([3,2,1,1,1])
sage: parent(x)
Partitions
sage: v = vector(RDF, [1,2,3])
sage: parent(v)
Vector space of dimension 3 over Real Double Field
```

The following are not considered to be elements, so the type is returned:

```
sage: d = int(42)  # Python int
sage: parent(d)
<type 'int'>
sage: L = range(10)
sage: parent(L)
<type 'list'>
```

sage.structure.element.**set_coercion_model**(*cm*)

CHAPTER

TEN

UNIQUE REPRESENTATION

Abstract classes for cached and unique representation behavior.

See also:

`sage.structure.factory.UniqueFactory`

AUTHORS:

- Nicolas M. Thiery (2008): Original version.
- Simon A. King (2013-02): Separate cached and unique representation.
- Simon A. King (2013-08): Extended documentation.

10.1 What is a cached representation?

Instances of a class have a *cached representation behavior* when several instances constructed with the same arguments share the same memory representation. For example, calling twice:

```
sage: G = SymmetricGroup(6)
sage: H = SymmetricGroup(6)
```

to create the symmetric group on six elements gives back the same object:

```
sage: G is H
True
```

This is a standard design pattern. Besides saving memory, it allows for sharing cached data (say representation theoretical information about a group). And of course a look-up in the cache is faster than the creation of a new object.

10.1.1 Implementing a cached representation

Sage provides two standard ways to create a cached representation: `CachedRepresentation` and `UniqueFactory`. Note that, in spite of its name, `UniqueFactory` does not ensure *unique* representation behaviour, which will be explained below.

Using `CachedRepresentation`

It is often very easy to use `CachedRepresentation`: One simply writes a Python class and adds `CachedRepresentation` to the list of base classes. If one does so, then the arguments used to create an instance of this class will by default also be used as keys for the cache:

```
sage: from sage.structure.unique_representation import CachedRepresentation
sage: class C(CachedRepresentation):
....:     def __init__(self, a, b=0):
....:         self.a = a
....:         self.b = b
....:     def __repr__(self):
....:         return "C(%s, %s)"%(self.a, self.b)
....:
sage: a = C(1)
sage: a is C(1)
True
```

In addition, pickling just works, provided that Python is able to look up the class. Hence, in the following two lines, we explicitly put the class into the `__main__` module. This is needed in doctests, but not in an interactive session:

```
sage: import __main__
sage: __main__.C = C
sage: loads(dumps(a)) is a
True
```

Often, this very easy approach is sufficient for applications. However, there are some pitfalls. Since the arguments are used for caching, all arguments must be hashable, i.e., must be valid as dictionary keys:

```
sage: C((1,2))
C((1, 2), 0)
sage: C([1,2])
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

In addition, equivalent ways of providing the arguments are *not* automatically normalised when forming the cache key, and hence different but equivalent arguments may yield distinct instances:

```
sage: C(1) is C(1,0)
False
sage: C(1) is C(a=1)
False
sage: repr(C(1)) == repr(C(a=1))
True
```

It should also be noted that the arguments are compared by equality, not by identity. This is often desired, but can imply subtle problems. For example, since `C(1)` already is in the cache, and since the unit elements in different finite fields are all equal to the integer one, we find:

```
sage: GF(5)(1) == 1 == GF(3)(1)
True
sage: C(1) is C(GF(3)(1)) is C(GF(5)(1))
True
```

But `C(2)` is not in the cache, and the number two is not equal in different finite fields (i. e., `GF(5)(2) == GF(3)(2)` returns as `False`), even though it is equal to the number two in the ring of integers (`GF(5)(2) == 2 == GF(3)(2)` returns as `True`; equality is not transitive when comparing elements of *distinct* algebraic structures!!). Hence, we have:

```
sage: GF(5)(2) == GF(3)(2)
False
sage: C(GF(3)(2)) is C(GF(5)(2))
False
```

Normalising the arguments

CachedRepresentation uses the metaclass ClasscallMetaclass. Its `__classcall__` method is a WeakCachedFunction. This function creates an instance of the given class using the given arguments, unless it finds the result in the cache. This has the following implications:

- The arguments must be valid dictionary keys (i.e., they must be hashable; see above).
- It is a weak cache, hence, if the user does not keep a reference to the resulting instance, then it may be removed from the cache during garbage collection.
- It is possible to preprocess the input arguments by implementing a `__classcall__` or a `__classcall_private__` method, but in order to benefit from caching, `CachedRepresentation.__classcall__()` should at some point be called.

Note: For technical reasons, it is needed that `__classcall__` respectively `__classcall_private__` are "static methods", i.e., they are callable objects that do not bind to an instance or class. For example, a `cached_function` can be used here, because it is callable, but does not bind to an instance or class, because it has no `__get__()` method. A usual Python function, however, has a `__get__()` method and would thus under normal circumstances bind to an instance or class, and thus the instance or class would be passed to the function as the first argument. To prevent a callable object from being bound to the instance or class, one can prepend the `@staticmethod` decorator to the definition; see `staticmethod`.

For more on Python's `__get__()` method, see: http://docs.python.org/2/howto/descriptor.html

---

Warning: If there is preprocessing, then the preprocessed arguments passed to `CachedRepresentation.__classcall__()` must be invariant under the preprocessing. That is to say, preprocessing the input arguments twice must have the same effect as preprocessing the input arguments only once. That is to say, the preprocessing must be idempotent.

The reason for this warning lies in the way pickling is implemented. If the preprocessed arguments are passed to `CachedRepresentation.__classcall__()`, then the resulting instance will store the *preprocessed* arguments in some attribute, and will use them for pickling. If the pickle is unpickled, then preprocessing is applied to the preprocessed arguments—and this second round of preprocessing must not change the arguments further, since otherwise a different instance would be created.

We illustrate the warning by an example. Imagine that one has instances that are created with an integer-valued argument, but only depend on the *square* of the argument. It would be a mistake to square the given argument during preprocessing:

```
sage: class WrongUsage(CachedRepresentation):
....:     @staticmethod
....:     def __classcall__(cls, n):
....:         return super(WrongUsage,cls).__classcall__(cls, n^2)
....:     def __init__(self, n):
....:         self.n = n
....:     def __repr__(self):
....:         return "Something(%d)"%self.n
....:
sage: import __main__
sage: __main__.WrongUsage = WrongUsage # This is only needed in doctests
sage: w = WrongUsage(3); w
Something(9)
sage: w._reduction
(<class '__main__.WrongUsage'>, (9,), {})
```

Indeed, the reduction data are obtained from the preprocessed argument. By consequence, if the resulting instance is pickled and unpickled, the argument gets squared *again*:

```
sage: loads(dumps(w))
Something(81)
```

Instead, the preprocessing should only take the absolute value of the given argument, while the squaring should happen inside of the `__init__` method, where it won't mess with the cache:

```
sage: class BetterUsage(CachedRepresentation):
....:     @staticmethod
....:     def __classcall__(cls, n):
....:         return super(BetterUsage, cls).__classcall__(cls, abs(n))
....:     def __init__(self, n):
....:         self.n = n^2
....:     def __repr__(self):
....:         return "SomethingElse(%d)"%self.n
....:
sage: __main__.BetterUsage = BetterUsage # This is only needed in doctests
sage: b = BetterUsage(3); b
SomethingElse(9)
sage: loads(dumps(b)) is b
True
sage: b is BetterUsage(-3)
True
```

In our next example, we create a cached representation class `C` that returns an instance of a sub-class `C1` or `C2` depending on the given arguments. This is implemented in a static `__classcall_private__` method of `C`, letting it choose the sub-class according to the given arguments. Since a `__classcall_private__` method will be ignored on sub-classes, the caching of `CachedRepresentation` is available to both `C1` and `C2`. But for illustration, we overload the static `__classcall__` method on `C2`, doing some argument preprocessing. We also create a sub-class `C2b` of `C2`, demonstrating that the `__classcall__` method is used on the sub-class (in contrast to a `__classcall_private__` method!).

```
sage: class C(CachedRepresentation):
....:     @staticmethod
....:     def __classcall_private__(cls, n, implementation=0):
....:         if not implementation:
....:             return C.__classcall__(cls, n)
....:         if implementation==1:
....:             return C1(n)
....:         if implementation>1:
....:             return C2(n,implementation)
....:     def __init__(self, n):
....:         self.n = n
....:     def __repr__(self):
....:         return "C(%d, 0)"%self.n
....:
sage: class C1(C):
....:     def __repr__(self):
....:         return "C1(%d)"%self.n
....:
sage: class C2(C):
....:     @staticmethod
....:     def __classcall__(cls, n, implementation=0):
....:         if implementation:
....:             return super(C2, cls).__classcall__(cls, (n,)*implementation)
....:         return super(C2, cls).__classcall__(cls, n)
```

```
....:     def __init__(self, t):
....:         self.t = t
....:     def __repr__(self):
....:         return "C2(%s)"%repr(self.t)
....:
sage: class C2b(C2):
....:     def __repr__(self):
....:         return "C2b(%s)"%repr(self.t)
....:
sage: __main__.C2 = C2      # not needed in an interactive session
sage: __main__.C2b = C2b
```

In the above example, C drops the argument implementation if it evaluates to False, and since the cached __classcall__ is called in this case, we have:

```
sage: C(1)
C(1, 0)
sage: C(1) is C(1,0)
True
sage: C(1) is C(1,0) is C(1,None) is C(1,[])
True
```

(Note that we were able to bypass the issue of arguments having to be hashable by catching the empty list [] during preprocessing in the __classcall_private__ method. Similarly, unhashable arguments can be made hashable – e. g., lists normalized to tuples – in the __classcall_private__ method before they are further delegated to __classcall__. See TCrystal for an example.)

If we call C1 directly or if we provide implementation=1 to C, we obtain an instance of C1. Since it uses the __classcall__ method inherited from CachedRepresentation, the resulting instances are cached:

```
sage: C1(2)
C1(2)
sage: C(2, implementation=1)
C1(2)
sage: C(2, implementation=1) is C1(2)
True
```

The class C2 preprocesses the input arguments. Instances can, again, be obtained directly or by calling C:

```
sage: C(1, implementation=3)
C2((1, 1, 1))
sage: C(1, implementation=3) is C2(1,3)
True
```

The argument preprocessing of C2 is inherited by C2b, since __classcall__ and not __classcall_private__ is used. Pickling works, since the preprocessing of arguments is idempotent:

```
sage: c2b = C2b(2,3); c2b
C2b((2, 2, 2))
sage: loads(dumps(c2b)) is c2b
True
```

Using UniqueFactory

For creating a cached representation using a factory, one has to

- create a class *separately* from the factory. This class **must** inherit from `object`. Its instances **must** allow attribute assignment.
- write a method `create_key` (or `create_key_and_extra_args`) that creates the cache key from the given arguments.
- write a method `create_object` that creates an instance of the class from a given cache key.
- create an instance of the factory with a name that allows to conclude where it is defined.

An example:

```
sage: class C(object):
....:     def __init__(self, t):
....:         self.t = t
....:     def __repr__(self):
....:         return "C%s"%repr(self.t)
....:
sage: from sage.structure.factory import UniqueFactory
sage: class MyFactory(UniqueFactory):
....:     def create_key(self, n, m=None):
....:         if isinstance(n, (tuple,list)) and m is None:
....:             return tuple(n)
....:         return (n,)*m
....:     def create_object(self, version, key, **extra_args):
....:         # We ignore version and extra_args
....:         return C(key)
....:
```

Now, we define an instance of the factory, stating that it can be found under the name `"F"` in the `__main__` module. By consequence, pickling works:

```
sage: F = MyFactory("__main__.F")
sage: __main__.F = F                # not needed in an interactive session
sage: loads(dumps(F)) is F
True
```

We can now create *cached* instances of `C` by calling the factory. The cache only takes into account the key computed with the method `create_key` that we provided. Hence, different given arguments may result in the same instance. Note that, again, the cache is weak, hence, the instance might be removed from the cache during garbage collection, unless an external reference is preserved.

```
sage: a = F(1, 2); a
C(1, 1)
sage: a is F((1,1))
True
```

If the class of the returned instances is a sub-class of `object`, and **if** the resulting instance allows attribute assignment, then pickling of the resulting instances is automatically provided for, and respects the cache.

```
sage: loads(dumps(a)) is a
True
```

This is because an attribute is stored that explains how the instance was created:

```
sage: a._factory_data
(<class '__main__.MyFactory'>, (...), (1, 1), {})
```

Note: If a class is used that does not inherit from `object` then unique pickling is *not* provided.

Caching is only available if the factory is called. If an instance of the class is directly created, then the cache is not used:

```
sage: C((1,1))
C(1, 1)
sage: C((1,1)) is a
False
```

10.1.2 Comparing the two ways of implementing a cached representation

In this sub-section, we discuss advantages and disadvantages of the two ways of implementing a cached representation, depending on the type of application.

Simplicity and transparency

In many cases, turning a class into a cached representation requires nothing more than adding `CachedRepresentation` to the list of base classes of this class. This is, of course, a very easy and convenient way. Writing a factory would involve a lot more work.

If preprocessing of the arguments is needed, then we have seen how to do this by a `__classcall_private__` or `__classcall__` method. But these are double underscore methods and hence, for example, invisible in the automatically created reference manual. Moreover, preprocessing *and* caching are implemented in the same method, which might be confusing. In a unique factory, these two tasks are cleanly implemented in two separate methods. With a factory, it is possible to create the resulting instance by arguments that are different from the key used for caching. This is significantly restricted with CachedRepresentation due to the requirement that argument preprocessing be idempotent.

Hence, if advanced preprocessing is needed, then `UniqueFactory` might be easier and more transparent to use than `CachedRepresentation`.

Class inheritance

Using `CachedRepresentation` has the advantage that one has a class and creates cached instances of this class by the usual Python syntax:

```
sage: G = SymmetricGroup(6)
sage: issubclass(SymmetricGroup, sage.structure.unique_representation.CachedRepresentation)
True
sage: isinstance(G, SymmetricGroup)
True
```

In contrast, a factory is just a callable object that returns something that has absolutely nothing to do with the factory, and may in fact return instances of quite different classes:

```
sage: isinstance(GF, sage.structure.factory.UniqueFactory)
True
sage: K5 = GF(5)
sage: type(K5)
<class 'sage.rings.finite_rings.finite_field_prime_modn.FiniteField_prime_modn_with_category'>
sage: K25 = GF(25, 'x')
sage: type(K25)
<class 'sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro_with_category'>
sage: Kp = GF(next_prime_power(1000000)^2, 'x')
```

```
sage: type(Kp)
<class 'sage.rings.finite_rings.finite_field_pari_ffelt.FiniteField_pari_ffelt_with_category'>
```

This can be confusing to the user. Namely, the user might determine the class of an instance and try to create further instances by calling the class rather than the factory—which is a mistake since it works around the cache (and also since the class might be more restrictive than the factory – i. e., the type of `K5` in the above doctest cannot be called on a prime power which is not a prime). This mistake can more easily be avoided by using `CachedRepresentation`.

We have seen above that one can easily create new cached-representation classes by subclassing an existing cached-representation class, even making use of an existing argument preprocess. This would be much more complicated with a factory. Namely, one would need to rewrite old factories making them aware of the new classes, and/or write new factories for the new classes.

Python versus extension classes

`CachedRepresentation` uses a metaclass, namely `ClasscallMetaclass`. Hence, it can currently not be a Cython extension class. Moreover, it is supposed to be used by providing it as a base class. But in typical applications, one also has another base class, say, `Parent`. Hence, one would like to create a class with at least two base classes, which is currently impossible in Cython extension classes.

In other words, when using `CachedRepresentation`, one must work with Python classes. These can be defined in Cython code (`.pyx` files) and can thus benefit from Cython's speed inside of their methods, but they must not be `cdef class` and can thus not use `cdef` attributes or methods.

Such restrictions do not exist when using a factory. However, if attribute assignment does not work, then the automatic pickling provided by `UniqueFactory` will not be available.

10.2 What is a unique representation?

Instances of a class have a *unique instance behavior* when instances of this class evaluate equal if and only if they are identical. Sage provides the base class `WithEqualityById`, which provides comparison by identity and a hash that is determined by the memory address of the instance. Both the equality test and the hash are implemented in Cython and are very fast, even when one has a Python class inheriting from `WithEqualityById`.

In many applications, one wants to combine unique instance and cached representation behaviour. This is called *unique representation* behaviour. We have seen above that symmetric groups have a *cached* representation behaviour. However, they do not show the *unique* representation behaviour, since they are equal to groups created in a totally different way, namely to subgroups:

```
sage: G = SymmetricGroup(6)
sage: G3 = G.subgroup([G((1,2,3,4,5,6)),G((1,2))])
sage: G is G3
False
sage: type(G) == type(G3)
False
sage: G == G3
True
```

The unique representation behaviour can conveniently be implemented with a class that inherits from `UniqueRepresentation`: By adding `UniqueRepresentation` to the base classes, the class will simultaneously inherit from `CachedRepresentation` and from `WithEqualityById`.

For example, a symmetric function algebra is uniquely determined by the base ring. Thus, it is reasonable to use `UniqueRepresentation` in this case:

```
sage: isinstance(SymmetricFunctions(CC), SymmetricFunctions)
True
sage: issubclass(SymmetricFunctions, UniqueRepresentation)
True
```

`UniqueRepresentation` differs from `CachedRepresentation` only by adding `WithEqualityById` as a base class. Hence, the above examples of argument preprocessing work for `UniqueRepresentation` as well.

Note that a cached representation created with `UniqueFactory` does *not* automatically provide unique representation behaviour, in spite of its name! Hence, for unique representation behaviour, one has to implement hash and equality test accordingly, for example by inheriting from `WithEqualityById`.

class `sage.structure.unique_representation.`**`CachedRepresentation`**

Bases: `object`

Classes derived from CachedRepresentation inherit a weak cache for their instances.

Note: If this class is used as a base class, then instances are (weakly) cached, according to the arguments used to create the instance. Pickling is provided, of course by using the cache.

Note: Using this class, one can have arbitrary hash and comparison. Hence, *unique* representation behaviour is *not* provided.

See also:

`UniqueRepresentation`, `unique_representation`

EXAMPLES:

Providing a class with a weak cache for the instances is easy: Just inherit from `CachedRepresentation`:

```
sage: from sage.structure.unique_representation import CachedRepresentation
sage: class MyClass(CachedRepresentation):
....:     # all the rest as usual
....:     pass
```

We start with a simple class whose constructor takes a single value as argument (TODO: find a more meaningful example):

```
sage: class MyClass(CachedRepresentation):
....:     def __init__(self, value):
....:         self.value = value
....:     def __cmp__(self, other):
....:         c = cmp(type(self),type(other))
....:         if c: return c
....:         return cmp(self.value, other.value)
```

Two coexisting instances of `MyClass` created with the same argument data are guaranteed to share the same identity. Since trac ticket #12215, this is only the case if there is some strong reference to the returned instance, since otherwise it may be garbage collected:

```
sage: x = MyClass(1)
sage: y = MyClass(1)
sage: x is y               # There is a strong reference
True
sage: z = MyClass(2)
sage: x is z
False
```

In particular, modifying any one of them modifies the other (reference effect):

```
sage: x.value = 3
sage: x.value, y.value
(3, 3)
sage: y.value = 1
sage: x.value, y.value
(1, 1)
```

The arguments can consist of any combination of positional or keyword arguments, as taken by a usual `__init__` function. However, all values passed in should be hashable:

```
sage: MyClass(value = [1,2,3])
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

Argument preprocessing

Sometimes, one wants to do some preprocessing on the arguments, to put them in some canonical form. The following example illustrates how to achieve this; it takes as argument any iterable, and canonicalizes it into a tuple (which is hashable!):

```
sage: class MyClass2(CachedRepresentation):
....:     @staticmethod
....:     def __classcall__(cls, iterable):
....:         t = tuple(iterable)
....:         return super(MyClass2, cls).__classcall__(cls, t)
....:
....:     def __init__(self, value):
....:         self.value = value
....:
sage: x = MyClass2([1,2,3])
sage: y = MyClass2(tuple([1,2,3]))
sage: z = MyClass2(i for i in [1,2,3])
sage: x.value
(1, 2, 3)
sage: x is y, y is z
(True, True)
```

A similar situation arises when the constructor accepts default values for some of its parameters. Alas, the obvious implementation does not work:

```
sage: class MyClass3(CachedRepresentation):
....:     def __init__(self, value = 3):
....:         self.value = value
....:
sage: MyClass3(3) is MyClass3()
False
```

Instead, one should do:

```
sage: class MyClass3(UniqueRepresentation):
....:     @staticmethod
....:     def __classcall__(cls, value = 3):
....:         return super(MyClass3, cls).__classcall__(cls, value)
....:
....:     def __init__(self, value):
....:         self.value = value
```

```
....:
sage: MyClass3(3) is MyClass3()
True
```

A bit of explanation is in order. First, the call MyClass2([1,2,3]) triggers a call to MyClass2.__classcall__(MyClass2, [1,2,3]). This is an extension of the standard Python behavior, needed by CachedRepresentation, and implemented by the ClasscallMetaclass. Then, MyClass2.__classcall__ does the desired transformations on the arguments. Finally, it uses super to call the default implementation of __classcall__ provided by CachedRepresentation. This one in turn handles the caching and, if needed, constructs and initializes a new object in the class using __new__ and __init__ as usual.

Constraints:

- __classcall__() is a staticmethod (like, implicitly, __new__)
- the preprocessing on the arguments should be idempotent. That is, if MyClass2.__classcall__(<arguments>) calls CachedRepresentation.__classcall__(<preproce then MyClass2.__classcall__(<preprocessed_arguments>) should also result in a call to CachedRepresentation.__classcall__(<preprocessed_arguments>).
- MyClass2.__classcall__ should return the result of CachedRepresentation.__classcall__() without modifying it.

Other than that MyClass2.__classcall__ may play any tricks, like acting as a factory and returning objects from other classes.

Warning: It is possible, but strongly discouraged, to let the __classcall__ method of a class C return objects that are not instances of C. Of course, instances of a *subclass* of C are fine. Compare the examples in unique_representation.

We illustrate what is meant by an "idempotent" preprocessing. Imagine that one has instances that are created with an integer-valued argument, but only depend on the *square* of the argument. It would be a mistake to square the given argument during preprocessing:

```
sage: class WrongUsage(CachedRepresentation):
....:     @staticmethod
....:     def __classcall__(cls, n):
....:         return super(WrongUsage,cls).__classcall__(cls, n^2)
....:     def __init__(self, n):
....:         self.n = n
....:     def __repr__(self):
....:         return "Something(%d)"%self.n
....:
sage: import __main__
sage: __main__.WrongUsage = WrongUsage # This is only needed in doctests
sage: w = WrongUsage(3); w
Something(9)
sage: w._reduction
(<class '__main__.WrongUsage'>, (9,), {})
```

Indeed, the reduction data are obtained from the preprocessed arguments. By consequence, if the resulting instance is pickled and unpickled, the argument gets squared *again*:

```
sage: loads(dumps(w))
Something(81)
```

Instead, the preprocessing should only take the absolute value of the given argument, while the squaring should happen inside of the __init__ method, where it won't mess with the cache:

```
sage: class BetterUsage(CachedRepresentation):
....:     @staticmethod
....:     def __classcall__(cls, n):
....:         return super(BetterUsage, cls).__classcall__(cls, abs(n))
....:     def __init__(self, n):
....:         self.n = n^2
....:     def __repr__(self):
....:         return "SomethingElse(%d)"%self.n
....:
sage: __main__.BetterUsage = BetterUsage # This is only needed in doctests
sage: b = BetterUsage(3); b
SomethingElse(9)
sage: loads(dumps(b)) is b
True
sage: b is BetterUsage(-3)
True
```

Cached representation and mutability

`CachedRepresentation` is primarily intended for implementing objects which are (at least semantically) immutable. This is in particular assumed by the default implementations of `copy` and `deepcopy`:

```
sage: copy(x) is x
True
sage: from copy import deepcopy
sage: deepcopy(x) is x
True
```

However, in contrast to `UniqueRepresentation`, using `CachedRepresentation` allows for a comparison that is not by identity:

```
sage: t = MyClass(3)
sage: z = MyClass(2)
sage: t.value = 2
```

Now `t` and `z` are non-identical, but equal:

```
sage: t.value == z.value
True
sage: t == z
True
sage: t is z
False
```

More on cached representation and identity

`CachedRepresentation` is implemented by means of a cache. This cache uses weak references. Hence, when all other references to, say, `MyClass(1)` have been deleted, the instance is actually deleted from memory. A later call to `MyClass(1)` reconstructs the instance from scratch.

```
sage: class SomeClass(UniqueRepresentation):
....:     def __init__(self, i):
....:         print "creating new instance for argument %s"%i
....:         self.i = i
....:     def __del__(self):
....:         print "deleting instance for argument %s"%self.i
```

```
....:
sage: O = SomeClass(1)
creating new instance for argument 1
sage: O is SomeClass(1)
True
sage: O is SomeClass(2)
creating new instance for argument 2
deleting instance for argument 2
False
sage: del O
deleting instance for argument 1
sage: O = SomeClass(1)
creating new instance for argument 1
sage: del O
deleting instance for argument 1
```

Cached representation and pickling

The default Python pickling implementation (by reconstructing an object from its class and dictionary, see "The pickle protocol" in the Python Library Reference) does not preserve cached representation, as Python has no chance to know whether and where the same object already exists.

`CachedRepresentation` tries to ensure appropriate pickling by implementing a `__reduce__` method returning the arguments passed to the constructor:

```
sage: import __main__             # Fake MyClass being defined in a python module
sage: __main__.MyClass = MyClass
sage: x = MyClass(1)
sage: loads(dumps(x)) is x
True
```

`CachedRepresentation` uses the `__reduce__` pickle protocol rather than `__getnewargs__` because the latter does not handle keyword arguments:

```
sage: x = MyClass(value = 1)
sage: x.__reduce__()
(<function unreduce at ...>, (<class '__main__.MyClass'>, (), {'value': 1}))
sage: x is loads(dumps(x))
True
```

Note: The default implementation of `__reduce__` in `CachedRepresentation` requires to store the constructor's arguments in the instance dictionary upon construction:

```
sage: x.__dict__
{'_reduction': (<class '__main__.MyClass'>, (), {'value': 1}), 'value': 1}
```

It is often easy in a derived subclass to reconstruct the constructor's arguments from the instance data structure. When this is the case, `__reduce__` should be overridden; automagically the arguments won't be stored anymore:

```
sage: class MyClass3(UniqueRepresentation):
....:     def __init__(self, value):
....:         self.value = value
....:
....:     def __reduce__(self):
....:         return (MyClass3, (self.value,))
....:
sage: import __main__; __main__.MyClass3 = MyClass3  # Fake MyClass3 being defined in a python m
```

```
sage: x = MyClass3(1)
sage: loads(dumps(x)) is x
True
sage: x.__dict__
{'value': 1}
```

Migrating classes to `CachedRepresentation` and unpickling

We check that, when migrating a class to `CachedRepresentation`, older pickles can still be reasonably unpickled. Let us create a (new style) class, and pickle one of its instances:

```
sage: class MyClass4(object):
....:     def __init__(self, value):
....:         self.value = value
....:
sage: import __main__; __main__.MyClass4 = MyClass4  # Fake MyClass4 being defined in a python m
sage: pickle = dumps(MyClass4(1))
```

It can be unpickled:

```
sage: y = loads(pickle)
sage: y.value
1
```

Now, we upgrade the class to derive from `UniqueRepresentation`, which inherits from `CachedRepresentation`:

```
sage: class MyClass4(UniqueRepresentation, object):
....:     def __init__(self, value):
....:         self.value = value
sage: import __main__; __main__.MyClass4 = MyClass4  # Fake MyClass4 being defined in a python m
sage: __main__.MyClass4 = MyClass4
```

The pickle can still be unpickled:

```
sage: y = loads(pickle)
sage: y.value
1
```

Note however that, for the reasons explained above, unique representation is not guaranteed in this case:

```
sage: y is MyClass4(1)
False
```

Todo

Illustrate how this can be fixed on a case by case basis.

Now, we redo the same test for a class deriving from SageObject:

```
sage: class MyClass4(SageObject):
....:     def __init__(self, value):
....:         self.value = value
sage: import __main__; __main__.MyClass4 = MyClass4  # Fake MyClass4 being defined in a python m
sage: pickle = dumps(MyClass4(1))

sage: class MyClass4(UniqueRepresentation, SageObject):
```

```
....:     def __init__(self, value):
....:         self.value = value
sage: __main__.MyClass4 = MyClass4
sage: y = loads(pickle)
sage: y.value
1
```

Caveat: unpickling instances of a formerly old-style class is not supported yet by default:

```
sage: class MyClass4:
....:     def __init__(self, value):
....:         self.value = value
sage: import __main__; __main__.MyClass4 = MyClass4  # Fake MyClass4 being defined in a python m
sage: pickle = dumps(MyClass4(1))

sage: class MyClass4(UniqueRepresentation, SageObject):
....:     def __init__(self, value):
....:         self.value = value
sage: __main__.MyClass4 = MyClass4
sage: y = loads(pickle)  # todo: not implemented
sage: y.value            # todo: not implemented
1
```

Rationale for the current implementation

CachedRepresentation and derived classes use the ClasscallMetaclass of the standard Python type. The following example explains why.

We define a variant of MyClass where the calls to __init__ are traced:

```
sage: class MyClass(CachedRepresentation):
....:     def __init__(self, value):
....:         print "initializing object"
....:         self.value = value
....:
```

Let us create an object twice:

```
sage: x = MyClass(1)
initializing object
sage: z = MyClass(1)
```

As desired the __init__ method was only called the first time, which is an important feature.

As far as we can tell, this is not achievable while just using __new__ and __init__ (as defined by type; see Section Basic Customization in the Python Reference Manual). Indeed, __init__ is called systematically on the result of __new__ whenever the result is an instance of the class.

Another difficulty is that argument preprocessing (as in the example above) cannot be handled by __new__, since the unprocessed arguments will be passed down to __init__.

class sage.structure.unique_representation.**UniqueRepresentation**

Bases: sage.structure.unique_representation.CachedRepresentation, sage.misc.fast_methods.WithEqualityById

Classes derived from UniqueRepresentation inherit a unique representation behavior for their instances.

See also:

unique_representation

EXAMPLES:

The short story: to construct a class whose instances have a unique representation behavior one just has to do:

```
sage: class MyClass(UniqueRepresentation):
....:     # all the rest as usual
....:     pass
```

Everything below is for the curious or for advanced usage.

What is unique representation?

Instances of a class have a *unique representation behavior* when instances evaluate equal if and only if they are identical (i.e., share the same memory representation), if and only if they were created using equal arguments. For example, calling twice:

```
sage: f = SymmetricFunctions(QQ)
sage: g = SymmetricFunctions(QQ)
```

to create the symmetric function algebra over $\mathbf{Q}$ actually gives back the same object:

```
sage: f == g
True
sage: f is g
True
```

This is a standard design pattern. It allows for sharing cached data (say representation theoretical information about a group) as well as for very fast hashing and equality testing. This behaviour is typically desirable for parents and categories. It can also be useful for intensive computations where one wants to cache all the operations on a small set of elements (say the multiplication table of a small group), and access this cache as quickly as possible.

`UniqueRepresentation` is very easy to use: a class just needs to derive from it, or make sure some of its super classes does. Also, it groups together the class and the factory in a single gadget:

```
sage: isinstance(SymmetricFunctions(CC), SymmetricFunctions)
True
sage: issubclass(SymmetricFunctions, UniqueRepresentation)
True
```

This nice behaviour is not available when one just uses a factory:

```
sage: isinstance(GF(7), GF)
Traceback (most recent call last):
...
TypeError: isinstance() arg 2 must be a class, type, or tuple of classes and types
sage: isinstance(GF, sage.structure.factory.UniqueFactory)
True
```

In addition, `UniqueFactory` only provides the *cached* representation behaviour, but not the *unique* representation behaviour—the examples in `unique_representation` explain this difference.

On the other hand, the `UniqueRepresentation` class is more intrusive, as it imposes a behavior (and a metaclass) on all the subclasses. In particular, the unique representation behaviour is imposed on *all* subclasses (unless the `__classcall__` method is overloaded and not called in the subclass, which is not recommended). Its implementation is also more technical, which leads to some subtleties.

EXAMPLES:

We start with a simple class whose constructor takes a single value as argument. This pattern is similar to what is done in sage.combinat.sf.sf.SymmetricFunctions:

```
sage: class MyClass(UniqueRepresentation):
....:     def __init__(self, value):
....:         self.value = value
....:     def __cmp__(self, other):
....:         c = cmp(type(self),type(other))
....:         if c: return c
....:         print "custom cmp"
....:         return cmp(self.value, other.value)
....:
```

Two coexisting instances of MyClass created with the same argument data are guaranteed to share the same identity. Since trac ticket #12215, this is only the case if there is some strong reference to the returned instance, since otherwise it may be garbage collected:

```
sage: x = MyClass(1)
sage: y = MyClass(1)
sage: x is y               # There is a strong reference
True
sage: z = MyClass(2)
sage: x is z
False
```

In particular, modifying any one of them modifies the other (reference effect):

```
sage: x.value = 3
sage: x.value, y.value
(3, 3)
sage: y.value = 1
sage: x.value, y.value
(1, 1)
```

Rich comparison by identity is used when possible (hence, for ==, for !=, and for identical arguments in the case of <, <=, >= and >), which is as fast as it can get. Only if identity is not enough to decide the answer of a comparison, the custom comparison is called:

```
sage: x == y
True
sage: z = MyClass(2)
sage: x == z, x is z
(False, False)
sage: x <= x
True
sage: x != z
True
sage: x <= z
custom cmp
True
sage: x > z
custom cmp
False
```

A hash function equivalent to object.__hash__() is used, which is compatible with comparison by identity. However this means that the hash function may change in between Sage sessions, or even within the same Sage session.

```
sage: hash(x) == object.__hash__(x)
True
```

Warning: It is possible to inherit from `UniqueRepresentation` and then overload comparison in a way that destroys the unique representation property. We strongly recommend against it! You should use `CachedRepresentation` instead.

Mixing super types and super classes

TESTS:

For the record, this test did fail with previous implementation attempts:

```
sage: class bla(UniqueRepresentation, SageObject):
....:     pass
....:
sage: b = bla()
```

`sage.structure.unique_representation.`**`unreduce`**(*cls*, *args*, *keywords*)

Calls a class on the given arguments:

```
sage: sage.structure.unique_representation.unreduce(Integer, (1,), {})
1
```

Todo

should reuse something preexisting ...

CHAPTER

ELEVEN

FACTORY FOR CACHED REPRESENTATIONS

See also:

`sage.structure.unique_representation`

Using a `UniqueFactory` is one way of implementing a *cached representation behaviour*. In spite of its name, using a `UniqueFactory` is not enough to ensure the *unique representation behaviour*. See `unique_representation` for a detailed explanation.

With a `UniqueFactory`, one can preprocess the given arguments. There is special support for specifying a subset of the arguments that serve as the unique key, so that still *all* given arguments are used to create a new instance, but only the specified subset is used to look up in the cache. Typically, this is used to construct objects that accept an optional `check=[True|False]` argument, but whose result should be unique regardless of said optional argument. (This use case should be handled with care, though: Any checking which isn't done in the `create_key` or `create_key_and_extra_args` method will be done only when a new object is generated, but not when a cached object is retrieved from cache. Consequently, if the factory is once called with `check=False`, a subsequent call with `check=True` cannot be expected to perform all checks unless these checks are all in the `create_key` or `create_key_and_extra_args` method.)

For a class derived from `CachedRepresentation`, argument preprocessing can be obtained by providing a custom static `__classcall__` or `__classcall_private__` method, but this seems less transparent. When argument preprocessing is not needed or the preprocess is not very sophisticated, then generally `CachedRepresentation` is much easier to use than a factory.

AUTHORS:

- Robert Bradshaw (2008): initial version.
- Simon King (2013): extended documentation.
- Julian Rueth (2014-05-09): use `_cache_key` if parameters are unhashable

class `sage.structure.factory.`**`UniqueFactory`**

Bases: `sage.structure.sage_object.SageObject`

This class is intended to make it easy to cache objects.

It is based on the idea that the object is uniquely defined by a set of defining data (the key). There is also the possibility of some non-defining data (extra args) which will be used in initial creation, but not affect the caching.

> **Warning:** This class only provides *cached representation behaviour*. Hence, using `UniqueFactory`, it is still possible to create distinct objects that evaluate equal. Unique representation behaviour can be added, for example, by additionally inheriting from `sage.misc.fast_methods.WithEqualityById`.

The objects created are cached (using weakrefs) based on their key and returned directly rather than re-created if requested again. Pickling is taken care of by the factory, and will return the same object for the same version of Sage, and distinct (but hopefully equal) objects for different versions of Sage.

> **Warning:** The objects returned by a `UniqueFactory` must be instances of new style classes (hence, they must be instances of `object`) that must not only allow a weak reference, but must accept general attribute assignment. Otherwise, pickling won't work.

USAGE:

A *unique factory* provides a way to create objects from parameters (the type of these objects can depend on the parameters, and is often determined only at runtime) and to cache them by a certain key derived from these parameters, so that when the factory is being called again with the same parameters (or just with parameters which yield the same key), the object is being returned from cache rather than constructed anew.

An implementation of a unique factory consists of a factory class and an instance of this factory class.

The factory class has to be a class inheriting from `UniqueFactory`. Typically it only needs to implement `create_key()` (a method that creates a key from the given parameters, under which key the object will be stored in the cache) and `create_object()` (a method that returns the actual object from the key). Sometimes, one would also implement `create_key_and_extra_args()` (this differs from `create_key()` in allowing to also create some additional arguments from the given parameters, which arguments then get passed to `create_object()` and thus can have an effect on the initial creation of the object, but do *not* affect the key) or `other_keys()`. Other methods are not supposed to be overloaded.

The factory class itself cannot be called to create objects. Instead, an instance of the factory class has to be created first. For technical reasons, this instance has to be provided with a name that allows Sage to find its definition. Specifically, the name of the factory instance (or the full path to it, if it is not in the global namespace) has to be passed to the factory class as a string variable. So, if our factory class has been called `A` and is located in `sage/spam/battletoads.py`, then we need to define an instance (say, `B`) of `A` by writing `B = A("sage.spam.battletoads.B")` (or `B = A("B")` if this `B` will be imported into global namespace). This instance can then be used to create objects (by calling `B(*parameters)`).

Notice that the objects created by the factory don't inherit from the factory class. They *do* know about the factory that created them (this information, along with the keys under which this factory caches them, is stored in the `_factory_data` attributes of the objects), but not via inheritance.

EXAMPLES:

The below examples are rather artificial and illustrate particular aspects. For a "real-life" usage case of `UniqueFactory`, see the finite field factory in `sage.rings.finite_rings.constructor`.

In many cases, a factory class is implemented by providing the two methods `create_key()` and `create_object()`. In our example, we want to demonstrate how to use "extra arguments" to choose a specific implementation, with preference given to an instance found in the cache, even if its implementation is different. Hence, we implement `create_key_and_extra_args()` rather than `create_key()`, putting the chosen implementation into the extra arguments. Then, in the `create_object()` method, we create and return instances of the specified implementation.

```
sage: from sage.structure.factory import UniqueFactory
sage: class MyFactory(UniqueFactory):
....:     def create_key_and_extra_args(self, *args, **kwds):
....:         return args, {'impl':kwds.get('impl', None)}
....:     def create_object(self, version, key, **extra_args):
....:         impl = extra_args['impl']
....:         if impl=='C':
....:             return C(*key)
....:         if impl=='D':
....:             return D(*key)
```

```
....:         return E(*key)
....:
```

Now we can create a factory instance. It is supposed to be found under the name `"F"` in the `"__main__"` module. Note that in an interactive session, `F` would automatically be in the `__main__` module. Hence, the second and third of the following four lines are only needed in doctests.

```
sage: F = MyFactory("__main__.F")
sage: import __main__
sage: __main__.F = F
sage: loads(dumps(F)) is F
True
```

Now we create three classes `C`, `D` and `E`. The first is a Cython extension-type class that does not allow weak references nor attribute assignment. The second is a Python class that is not derived from `object`. The third allows attribute assignment and is derived from `object`.

```
sage: cython("cdef class C: pass")
sage: class D:
....:     def __init__(self, *args):
....:         self.t = args
....:     def __repr__(self):
....:         return "D%s"%repr(self.t)
....:
sage: class E(D, object): pass
```

Again, being in a doctest, we need to put the class `D` into the `__main__` module, so that Python can find it:

```
sage: import __main__
sage: __main__.D = D
```

It is impossible to create an instance of `C` with our factory, since it does not allow weak references:

```
sage: F(1, impl='C')
Traceback (most recent call last):
...
TypeError: cannot create weak reference to '....C' object
```

Let us try again, with a Cython class that does allow weak references. Now, creation of an instance using the factory works:

```
sage: cython('''cdef class C:
....:     cdef __weakref__
....: ''')
....:
sage: c = F(1, impl='C')
sage: isinstance(c, C)
True
```

The cache is used when calling the factory again—even if it is suggested to use a different implementation. This is because the implementation is only considered an "extra argument" that does not count for the key.

```
sage: c is F(1, impl='C') is F(1, impl="D") is F(1)
True
```

However, pickling and unpickling does not use the cache. This is because the factory has tried to assign an attribute to the instance that provides information on the key used to create the instance, but failed:

```
sage: loads(dumps(c)) is c
False
```

```
sage: hasattr(c, '_factory_data')
False
```

We have already seen that our factory will only take the requested implementation into account if the arguments used as key have not been used yet. So, we use other arguments to create an instance of class `D`:

```
sage: d = F(2, impl='D')
sage: isinstance(d, D)
True
```

The factory only knows about the pickling protocol used by new style classes. Hence, again, pickling and unpickling fails to use the cache, even though the "factory data" are now available:

```
sage: loads(dumps(d)) is d
False
sage: d._factory_data
(<class '__main__.MyFactory'>, (...), (2,), {'impl': 'D'})
```

Only when we have a new style class that can be weak referenced and allows for attribute assignment, everything works:

```
sage: e = F(3)
sage: isinstance(e, E)
True
sage: loads(dumps(e)) is e
True
sage: e._factory_data
(<class '__main__.MyFactory'>, (...), (3,), {'impl': None})
```

create_key (**args, **kwds*)

Given the parameters (arguments and keywords), create a key that uniquely determines this object.

EXAMPLES:

```
sage: from sage.structure.test_factory import test_factory
sage: test_factory.create_key(1, 2, key=5)
(1, 2)
```

create_key_and_extra_args (**args, **kwds*)

Return a tuple containing the key (uniquely defining data) and any extra arguments (empty by default).

Defaults to `create_key()`.

EXAMPLES:

```
sage: from sage.structure.test_factory import test_factory
sage: test_factory.create_key_and_extra_args(1, 2, key=5)
((1, 2), {})
sage: GF.create_key_and_extra_args(3, foo='value')
((3, ('x',), None, 'modn', "{'foo': 'value'}", 3, 1, True), {'foo': 'value'})
```

create_object (*version, key, **extra_args*)

Create the object from the key and extra arguments. This is only called if the object was not found in the cache.

EXAMPLES:

```
sage: from sage.structure.test_factory import test_factory
sage: test_factory.create_object(0, (1,2,3))
Making object (1, 2, 3)
<sage.structure.test_factory.A instance at ...>
```

```
sage: test_factory('a')
Making object ('a',)
<sage.structure.test_factory.A instance at ...>
sage: test_factory('a') # NOT called again
<sage.structure.test_factory.A instance at ...>
```

get_object (*version*, *key*, *extra_args*)

Returns the object corresponding to `key`, creating it with `extra_args` if necessary (for example, it isn't in the cache or it is unpickling from an older version of Sage).

EXAMPLES:

```
sage: from sage.structure.test_factory import test_factory
sage: a = test_factory.get_object(3.0, 'a', {}); a
Making object a
<sage.structure.test_factory.A instance at ...>
sage: test_factory.get_object(3.0, 'a', {}) is test_factory.get_object(3.0, 'a', {})
True
sage: test_factory.get_object(3.0, 'a', {}) is test_factory.get_object(3.1, 'a', {})
Making object a
False
sage: test_factory.get_object(3.0, 'a', {}) is test_factory.get_object(3.0, 'b', {})
Making object b
False
```

TESTS:

Check that trac ticket #16317 has been fixed, i.e., caching works for unhashable objects:

```
sage: K.<u> = Qq(4)
sage: test_factory.get_object(3.0, (K(1), 'c'), {})  is test_factory.get_object(3.0, (K(1),
Making object (1 + O(2^20), 'c')
True
```

get_version (*sage_version*)

This is provided to allow more or less granular control over pickle versioning. Objects pickled in the same version of Sage will unpickle to the same rather than simply equal objects. This can provide significant gains as arithmetic must be performed on objects with identical parents. However, if there has been an incompatible change (e.g. in element representation) we want the version number to change so coercion is forced between the two parents.

Defaults to the Sage version that is passed in, but coarser granularity can be provided.

EXAMPLES:

```
sage: from sage.structure.test_factory import test_factory
sage: test_factory.get_version((3,1,0))
(3, 1, 0)
```

other_keys (*key*, *obj*)

Sometimes during object creation, certain defaults are chosen which may result in a new (more specific) key. This allows the more specific key to be regarded as equivalent to the original key returned by `create_key()` for the purpose of lookup in the cache, and is used for pickling.

EXAMPLES:

The `GF` factory used to have a custom `other_keys()` method, but this was removed in trac ticket #16934:

```
sage: key, _ = GF.create_key_and_extra_args(27, 'k'); key
(27, ('k',), x^3 + 2*x + 1, 'givaro', '{}', 3, 3, True)
```

```
sage: K = GF.create_object(0, key); K
Finite Field in k of size 3^3
sage: GF.other_keys(key, K)
[]

sage: K = GF(7^40, 'a')
sage: loads(dumps(K)) is K
True
```

reduce_data (*obj*)

The results of this function can be returned from `__reduce__()`. This is here so the factory internals can change without having to re-write `__reduce__()` methods that use it.

EXAMPLE:

```
sage: V = FreeModule(ZZ, 5)
sage: factory, data = FreeModule.reduce_data(V)
sage: factory(*data)
Ambient free module of rank 5 over the principal ideal domain Integer Ring
sage: factory(*data) is V
True

sage: from sage.structure.test_factory import test_factory
sage: a = test_factory(1, 2)
Making object (1, 2)
sage: test_factory.reduce_data(a)
(<built-in function generic_factory_unpickle>,
 (<class 'sage.structure.test_factory.UniqueFactoryTester'>,
  (...),
  (1, 2),
  {}))
```

Note that the ellipsis `(...)` here stands for the Sage version.

`sage.structure.factory.`**generic_factory_reduce** (*self*, *proto*)

Used to provide a `__reduce__` method if one does not already exist.

EXAMPLES:

```
sage: V = QQ^6
sage: sage.structure.factory.generic_factory_reduce(V, 1) == V.__reduce_ex__(1)
True
```

`sage.structure.factory.`**generic_factory_unpickle** (*factory*, **args*)

Method used for unpickling the object.

The unpickling mechanism needs a plain Python function to call. It takes a factory as the first argument, passes the rest of the arguments onto the factory's `UniqueFactory.get_object()` method.

EXAMPLES:

```
sage: V = FreeModule(ZZ, 5)
sage: func, data = FreeModule.reduce_data(V)
sage: func is sage.structure.factory.generic_factory_unpickle
True
sage: sage.structure.factory.generic_factory_unpickle(*data) is V
True
```

TESTS:

The following was enabled in trac ticket #16349. Suppose we have defined (somewhere in the library of an old Sage version) a unique factory; in our example below, it returns polynomial rings. Now suppose that we want to replace the factory by something else, say, a class that provides the unique parent behaviour using UniqueRepresentation. We show here how to make it possible to unpickle a pickle created with the factory, automatically turning it into an instance of the new class.

First, we create the factory. In a doctest, it is needed to explicitly put it into `__main__`, so that it can be located when pickling. Also, it is needed that we work with a new-style class:

```
sage: from sage.structure.factory import UniqueFactory
sage: import __main__
sage: class OldStuff(object):
....:     def __init__(self, n, **extras):
....:         self.n = n
....:     def __repr__(self):
....:         return "Rotten old thing of level {}".format(self.n)
sage: __main__.OldStuff = OldStuff
sage: class MyFactory(UniqueFactory):
....:     def create_object(self, version, key, **extras):
....:         return OldStuff(key[0])
....:     def create_key(self, *args):
....:         return args
sage: F = MyFactory('__main__.F')
sage: __main__.F = F
sage: a = F(3); a
Rotten old thing of level 3
sage: loads(dumps(a)) is a
True
```

Now, we create a pickle (the string returned by `dumps(a)`):

```
sage: s = dumps(a)
```

We create a new class, derived from UniqueRepresentation, that shall replace the old factory. In particular, the class has to have the same name as the old factory, and has to be put into the same module (here: `__main__`). We turn it into a sub-class of the old class, but this is just to save the effort of writing a new init method:

```
sage: from sage.structure.unique_representation import UniqueRepresentation
sage: class F(UniqueRepresentation, OldStuff):
....:     def __repr__(self):
....:         return "Shiny new thing of level {}".format(self.n)
sage: __main__.F = F
```

The old pickle correctly unpickles as an instance of the new class, which is of course different from the instance of the old class, but exhibits unique object behaviour as well:

```
sage: b = loads(s); b
Shiny new thing of level 3
sage: a is b
False
sage: loads(dumps(b)) is b
True
```

`sage.structure.factory.`**`lookup_global`**`(`*name*`)`

Used in unpickling the factory itself.

EXAMPLES:

```
sage: from sage.structure.factory import lookup_global
sage: lookup_global('ZZ')
Integer Ring
```

```
sage: lookup_global('sage.rings.all.ZZ')
Integer Ring
```

`sage.structure.factory.`**`register_factory_unpickle`**(*name*, *callable*)

Register a callable to handle the unpickling from an old `UniqueFactory` object.

`UniqueFactory` pickles use a global name through `generic_factory_unpickle()`, so the usual `register_unpickle_override()` cannot be used here.

See also:

`generic_factory_unpickle()`

TESTS:

This is similar to the example given in `generic_factory_unpickle()`, but here we will use a function to explicitly return a polynomial ring.

First, we create the factory. In a doctest, it is needed to explicitly put it into `__main__`, so that it can be located when pickling. Also, it is needed that we work with a new-style class:

```
sage: from sage.structure.factory import UniqueFactory, register_factory_unpickle
sage: import __main__
sage: class OldStuff(object):
....:     def __init__(self, n, **extras):
....:         self.n = n
....:     def __repr__(self):
....:         return "Rotten old thing of level {}".format(self.n)
sage: __main__.OldStuff = OldStuff
sage: class MyFactory(UniqueFactory):
....:     def create_object(self, version, key, **extras):
....:         return OldStuff(key[0])
....:     def create_key(self, *args):
....:         return args
sage: G = MyFactory('__main__.G')
sage: __main__.G = G
sage: a = G(3); a
Rotten old thing of level 3
sage: loads(dumps(a)) is a
True
```

Now, we create a pickle (the string returned by `dumps(a)`):

```
sage: s = dumps(a)
```

We create the function which will handle the unpickling:

```
sage: def foo(n, **kwds):
....:     return PolynomialRing(QQ, n, 'x')
sage: register_factory_unpickle('__main__.G', foo)
```

The old pickle correctly unpickles as an explicit polynomial ring:

```
sage: loads(s)
Multivariate Polynomial Ring in x0, x1, x2 over Rational Field
```

CHAPTER

TWELVE

DYNAMIC CLASSES

Why dynamic classes?

The short answer:

- Multiple inheritance is a powerful tool for constructing new classes by combining preexisting building blocks.
- There is a combinatorial explosion in the number of potentially useful classes that can be produced this way.
- The implementation of standard mathematical constructions calls for producing such combinations automatically.
- Dynamic classes, i.e. classes created on the fly by the Python interpreter, are a natural mean to achieve this.

The long answer:

Say we want to construct a new class `MyPermutation` for permutations in a given set S (in Sage, S will be modelled by a parent, but we won't discuss this point here). First, we have to choose a data structure for the permutations, typically among the following:

- Stored by cycle type
- Stored by code
- Stored in list notation - C arrays of short ints (for small permutations) - python lists of ints (for huge permutations) - ...
- Stored by reduced word
- Stored as a function
- ...

Luckily, the Sage library provides (or will provide) classes implementing each of those data structures. Those classes all share a common interface (or possibly a common abstract base class). So we can just derive our class from the chosen one:

```
class MyPermutation(PermutationCycleType):
    ...
```

Then we may want to further choose a specific memory behavior (unique representation, copy-on-write) which (hopefuly) can again be achieved by inheritance:

```
class MyPermutation(UniqueRepresentation, PermutationCycleType):
     ...
```

Finaly, we may want to endow the permutations in S with further operations coming from the (algebraic) structure of S:

- group operations

- or just monoid operations (for a subset of permutations not stable by inverse)
- poset operations (for left/right/Bruhat order)
- word operations (searching for substrings, patterns, ...)

Or any combination thereof. Now, our class typically looks like:

```
class MyPermutation(UniqueRepresentation, PermutationCycleType, PosetElement, GroupElement):
    ...
```

Note the combinatorial explosion in the potential number of classes which can be created this way.

In practice, such classes will be used in mathematical constructions like:

```
SymmetricGroup(5).subset(... TODO: find a good example in the context above ...)
```

In such a construction, the structure of the result, and therefore the operations on its elements can only be determined at execution time. Let us take another standard construction:

```
A = cartesian_product( B, C )
```

Depending on the structure of B and C, and possibly on further options passed down by the user, A may be:

- an enumerated set
- a group
- an algebra
- a poset
- ...

Or any combination thereof.

Hardcoding classes for all potential combinations would be at best tedious. Furthermore, this would require a cumbersome mechanism to lookup the appropriate class depending on the desired combination.

Instead, one may use the ability of Python to create new classes dynamicaly:

```
type("class name", tuple of base classes, dictionary of methods)
```

This paradigm is powerful, but there are some technicalities to address. The purpose of this library is to standardize its use within Sage, and in particular to ensure that the constructed classes are reused whenever possible (unique representation), and can be pickled.

Combining dynamic classes and Cython classes

Cython classes cannot inherit from a dynamic class (there might be some partial support for this in the future). On the other hand, such an inheritance can be partially emulated using `__getattr__()`. See `sage.categories.examples.semigroups_cython` for an example.

class `sage.structure.dynamic_class.`**`DynamicClasscallMetaclass`**

Bases: `sage.structure.dynamic_class.DynamicMetaclass`, `sage.misc.classcall_metaclass.ClasscallMetaclass`

This invokes the nested_pickle on construction.

sage: from sage.misc.nested_class import NestedClassMetaclass sage: class A(object): ... __metaclass__ = NestedClassMetaclass ... class B(object): ... pass ... sage: A.B <class '__main__.A.B'> sage: getattr(sys.modules['__main__'], 'A.B', 'Not found') <class '__main__.A.B'>

class sage.structure.dynamic_class.**DynamicMetaclass**

Bases: type

A metaclass implementing an appropriate reduce-by-construction method

class sage.structure.dynamic_class.**TestClass**

A class used for checking that introspection works

bla()

bla ...

sage.structure.dynamic_class.**dynamic_class**(*name*, *bases*, *cls=None*, *reduction=None*, *doccls=None*, *prepend_cls_bases=True*, *cache=True*)

INPUT:

- name – a string
- bases – a tuple of classes
- cls – a class or None
- reduction – a tuple or None
- doccls – a class or None
- prepend_cls_bases – a boolean (default: True)
- cache – a boolean or "ignore_reduction" (default: True)

Constructs dynamically a new class C with name name, and bases bases. If cls is provided, then its methods will be inserted into C, and its bases will be prepended to bases (unless prepend_cls_bases is False).

The module, documentation and source instrospection is taken from doccls, or cls if doccls is None, or bases[0] if both are None (therefore bases should be non empty if cls` is ``None).

The constructed class can safely be pickled (assuming the arguments themselves can).

Unless cache is False, the result is cached, ensuring unique representation of dynamic classes.

See sage.structure.dynamic_class for a discussion of the dynamic classes paradigm, and its relevance to Sage.

EXAMPLES:

To setup the stage, we create a class Foo with some methods, cached methods, and lazy attributes, and a class Bar:

```
sage: from sage.misc.lazy_attribute import lazy_attribute
sage: from sage.misc.cachefunc import cached_function
sage: from sage.structure.dynamic_class import dynamic_class
sage: class Foo(object):
...       "The Foo class"
...       def __init__(self, x):
...           self._x = x
...       @cached_method
...       def f(self):
...           return self._x^2
...       def g(self):
...           return self._x^2
...       @lazy_attribute
...       def x(self):
...           return self._x
...
sage: class Bar:
```

```
...         def bar(self):
...             return self._x^2
...
```

We now create a class FooBar which is a copy of Foo, except that it also inherits from Bar:

```
sage: FooBar = dynamic_class("FooBar", (Bar,), Foo)
sage: x = FooBar(3)
sage: x.f()
9
sage: x.f() is x.f()
True
sage: x.x
3
sage: x.bar()
9
sage: FooBar.__name__
'FooBar'
sage: FooBar.__module__
'__main__'
```

```
sage: Foo.__bases__
(<type 'object'>,)
sage: FooBar.__bases__
(<type 'object'>, <class __main__.Bar at ...>)
sage: Foo.mro()
[<class '__main__.Foo'>, <type 'object'>]
sage: FooBar.mro()
[<class '__main__.FooBar'>, <type 'object'>, <class __main__.Bar at ...>]
```

Pickling

Dynamic classes are pickled by construction. Namely, upon unpickling, the class will be reconstructed by recalling dynamic_class with the same arguments:

```
sage: type(FooBar).__reduce__(FooBar)
(<function dynamic_class at ...>, ('FooBar', (<class __main__.Bar at ...>,), <class '__main__.Fo
```

Technically, this is achieved by using a metaclass, since the Python pickling protocol for classes is to pickle by name:

```
sage: type(FooBar)
<class 'sage.structure.dynamic_class.DynamicMetaclass'>
```

The following (meaningless) example illustrates how to customize the result of the reduction:

```
sage: BarFoo = dynamic_class("BarFoo", (Foo,), Bar, reduction = (str, (3,)))
sage: type(BarFoo).__reduce__(BarFoo)
(<type 'str'>, (3,))
sage: loads(dumps(BarFoo))
'3'
```

Caching

By default, the built class is cached:

```
sage: dynamic_class("FooBar", (Bar,), Foo) is FooBar
True
sage: dynamic_class("FooBar", (Bar,), Foo, cache=True) is FooBar
True
```

and the result depends on the reduction:

```
sage: dynamic_class("BarFoo", (Foo,), Bar, reduction = (str, (3,))) is BarFoo
True
sage: dynamic_class("BarFoo", (Foo,), Bar, reduction = (str, (2,))) is BarFoo
False
```

With `cache=False`, a new class is created each time:

```
sage: FooBar1 = dynamic_class("FooBar", (Bar,), Foo, cache=False); FooBar1
<class '__main__.FooBar'>
sage: FooBar2 = dynamic_class("FooBar", (Bar,), Foo, cache=False); FooBar2
<class '__main__.FooBar'>
sage: FooBar1 is FooBar
False
sage: FooBar2 is FooBar1
False
```

With `cache="ignore_reduction"`, the class does not depend on the reduction:

```
sage: BarFoo = dynamic_class("BarFoo", (Foo,), Bar, reduction = (str, (3,)), cache="ignore_reduc
sage: dynamic_class("BarFoo", (Foo,), Bar, reduction = (str, (2,)), cache="ignore_reduction") is
True
```

In particular, the reduction used is that provided upon creating the first class:

```
sage: dynamic_class("BarFoo", (Foo,), Bar, reduction = (str, (2,)), cache="ignore_reduction")._r
(<type 'str'>, (3,))
```

> **Warning:** The behaviour upon creating several dynamic classes from the same data but with different values for `cache` option is currently left unspecified. In other words, for a given application, it is recommended to consistently use the same value for that option.

TESTS:

```
sage: import __main__
sage: __main__.Foo = Foo
sage: __main__.Bar = Bar
sage: x = FooBar(3)
sage: x.__dict__      # Breaks without the __dict__ deletion in dynamic_class_internal
{'_x': 3}

sage: type(FooBar).__reduce__(FooBar)
(<function dynamic_class at ...>, ('FooBar', (<class __main__.Bar at ...>,), <class '__main__.Fo
sage: import cPickle
sage: cPickle.loads(cPickle.dumps(FooBar)) == FooBar
True
```

We check that instrospection works reasonably:

```
sage: sage.misc.sageinspect.sage_getdoc(FooBar)
'The Foo class\n'
```

Finally, we check that classes derived from UniqueRepresentation are handled gracefuly (despite them also using a metaclass):

```
sage: FooUnique = dynamic_class("Foo", (Bar, UniqueRepresentation))
sage: loads(dumps(FooUnique)) is FooUnique
True
```

sage.structure.dynamic_class.**dynamic_class_internal**(*name*, *bases*, *cls=None*, *reduction=None*, *doccls=None*, *prepend_cls_bases=True*)

See sage.structure.dynamic_class.dynamic_class? for indirect doctests.

TESTS:

```
sage: Foo1 = sage.structure.dynamic_class.dynamic_class_internal("Foo", (object,))
sage: Foo2 = sage.structure.dynamic_class.dynamic_class_internal("Foo", (object,), doccls = sage
sage: Foo3 = sage.structure.dynamic_class.dynamic_class_internal("Foo", (object,), cls    = sage
sage: all(Foo.__name__  == 'Foo'    for Foo in [Foo1, Foo2, Foo3])
True
sage: all(Foo.__bases__ == (object,) for Foo in [Foo1, Foo2, Foo3])
True
sage: Foo1.__module__ == object.__module__
True
sage: Foo2.__module__ == sage.structure.dynamic_class.TestClass.__module__
True
sage: Foo3.__module__ == sage.structure.dynamic_class.TestClass.__module__
True
sage: Foo1.__doc__ == object.__doc__
True
sage: Foo2.__doc__ == sage.structure.dynamic_class.TestClass.__doc__
True
sage: Foo3.__doc__ == sage.structure.dynamic_class.TestClass.__doc__
True
```

We check that instrospection works reasonably:

```
sage: import inspect
sage: inspect.getfile(Foo2)
'.../sage/structure/dynamic_class.pyc'
sage: inspect.getfile(Foo3)
'.../sage/structure/dynamic_class.pyc'
sage: sage.misc.sageinspect.sage_getsourcelines(Foo2)
(['class TestClass:...'], ...)
sage: sage.misc.sageinspect.sage_getsourcelines(Foo3)
(['class TestClass:...'], ...)
sage: sage.misc.sageinspect.sage_getsourcelines(Foo2())
(['class TestClass:...'], ...)
sage: sage.misc.sageinspect.sage_getsourcelines(Foo3())
(['class TestClass:...'], ...)
sage: sage.misc.sageinspect.sage_getsourcelines(Foo3().bla)
(['    def bla():...'], ...)
```

CHAPTER

THIRTEEN

ELEMENTS, ARRAY AND LISTS WITH CLONE PROTOCOL

This module defines several classes which are subclasses of `Element` and which roughly implement the "prototype" design pattern (see [Pro], [GOF]). Those classes are intended to be used to model *mathematical* objects, which are by essence immutable. However, in many occasions, one wants to construct the data-structure encoding of a new mathematical object by small modifications of the data structure encoding some already built object. For the resulting data-structure to correctly encode the mathematical object, some structural invariants must hold. One problem is that, in many cases, during the modification process, there is no possibility but to break the invariants.

For example, in a list modeling a permutation of $\{1, \ldots, n\}$, the integers must be distinct. A very common operation is to take a permutation to make a copy with some small modifications, like exchanging two consecutive values in the list or cycling some values. Though the result is clearly a permutations there's no way to avoid breaking the permutations invariants at some point during the modifications.

The main purpose of this module is to define the class

- `ClonableElement` as an abstract super class,

and its subclasses:

- `ClonableArray` for arrays (lists of fixed length) of objects;
- `ClonableList` for (resizable) lists of objects;
- `NormalizedClonableList` for lists of objects with a normalization method;
- `ClonableIntArray` for arrays of int.

See also:

The following parents from `sage.structure.list_clone_demo` demonstrate how to use them:

- `IncreasingArrays()` (see `IncreasingArray` and the parent class `IncreasingArrays`)
- `IncreasingLists()` (see `IncreasingList` and the parent class `IncreasingLists`)
- `SortedLists()` (see `SortedList` and the parent class `SortedLists`)
- `IncreasingIntArrays()` (see `IncreasingIntArray` and the parent class `IncreasingIntArrays`)

EXAMPLES:

We now demonstrate how `IncreasingArray` works, creating an instance `el` through its parent `IncreasingArrays()`:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: P = IncreasingArrays()
sage: P([1, 4 ,8])
[1, 4, 8]
```

If one tries to create this way a list which in not increasing, an error is raised:

```
sage: IncreasingArrays()([5, 4 ,8])
Traceback (most recent call last):
...
ValueError: array is not increasing
```

Once created modifying `el` is forbidden:

```
sage: el = P([1, 4 ,8])
sage: el[1] = 3
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

However, you can modify a temporarily mutable clone:

```
sage: with el.clone() as elc:
....:      elc[1] = 3
sage: [el, elc]
[[1, 4, 8], [1, 3, 8]]
```

We check that the original and the modified copy now are in a proper immutable state:

```
sage: el.is_immutable(), elc.is_immutable()
(True, True)
sage: elc[1] = 5
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

You can break the property that the list is increasing during the modification:

```
sage: with el.clone() as elc2:
....:     elc2[1] = 12
....:     print elc2
....:     elc2[2] = 25
[1, 12, 8]
sage: elc2
[1, 12, 25]
```

But this property must be restored at the end of the `with` block; otherwise an error is raised:

```
sage: with elc2.clone() as el3:
....:     el3[1] = 100
Traceback (most recent call last):
...
ValueError: array is not increasing
```

Finally, as an alternative to the `with` syntax one can use:

```
sage: el4 = copy(elc2)
sage: el4[1] = 10
sage: el4.set_immutable()
sage: el4.check()
```

REFERENCES:

AUTHORS:

- Florent Hivert (2010-03): initial revision

class `sage.structure.list_clone.`**`ClonableArray`**

Bases: `sage.structure.list_clone.ClonableElement`

Array with clone protocol

The class of objects which are `Element` behave as arrays (i.e. lists of fixed length) and implement the clone protocol. See `ClonableElement` for details about clone protocol.

INPUT:

- `parent` – a `Parent`
- `lst` – a list
- `check` – a boolean specifying if the invariant must be checked using method `check()`.
- `immutable` – a boolean telling wether the created element is immutable (defaults to `True`)

See also:

`IncreasingArray` for an example of usage.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: IA = IncreasingArrays()
sage: ia1 = IA([1, 4, 6]); ia1
[1, 4, 6]
sage: with ia1.clone() as ia2:
....:       ia2[1] = 5
sage: ia2
[1, 5, 6]
sage: with ia1.clone() as ia2:
....:       ia2[1] = 7
Traceback (most recent call last):
...
ValueError: array is not increasing
```

`check`()

Check that `self` fulfill the invariants

This is an abstract method. Subclasses are supposed to overload `check`.

EXAMPLES:

```
sage: from sage.structure.list_clone import ClonableArray
sage: ClonableArray(Parent(), [1,2,3]) # indirect doctest
Traceback (most recent call last):
...
NotImplementedError: this should never be called, please overload the check method
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: el = IncreasingArrays()([1,2,4]) # indirect doctest
```

`count`(*key*)

Returns number of `i`'s for which `s[i] == key`

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: c = IncreasingArrays()([1,2,2,4])
sage: c.count(1)
```

```
1
sage: c.count(2)
2
sage: c.count(3)
0
```

index (*x*, *start=None*, *stop=None*)

Returns the smallest `k` such that `s[k] == x` and `i <= k < j`

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: c = IncreasingArrays()([1,2,4])
sage: c.index(1)
0
sage: c.index(4)
2
sage: c.index(5)
Traceback (most recent call last):
...
ValueError: 5 is not in list
```

class `sage.structure.list_clone.`**`ClonableElement`**

Bases: `sage.structure.element.Element`

Abstract class for elements with clone protocol

This class is a subclass of `Element` and implements the "prototype" design pattern (see [Pro], [GOF]). The role of this class is:

- to manage copy and mutability and hashing of elements
- to ensure that at the end of a piece of code an object is restored in a meaningful mathematical state.

A class `C` inheriting from `ClonableElement` must implement the following two methods

- `obj.__copy__()` – returns a fresh copy of obj
- `obj.check()` – returns nothing, raise an exception if `obj` doesn't satisfies the data structure invariants

and ensure to call `obj._require_mutable()` at the beginning of any modifying method.

Additionally, one can also implement

- `obj._hash_()` – return the hash value of `obj`.

Then, given an instance `obj` of `C`, the following sequences of instructions ensures that the invariants of `new_obj` are properly restored at the end:

```
with obj.clone() as new_obj:
    ...
    # lot of invariant breaking modifications on new_obj
    ...
# invariants are ensured to be fulfilled
```

The following equivalent sequence of instructions can be used if speed is needed, in particular in Cython code:

```
new_obj = obj.__copy__()
...
# lot of invariant breaking modifications on new_obj
...
new_obj.set_immutable()
```

```
new_obj.check()
# invariants are ensured to be fulfilled
```

Finally, if the class implements the _hash_ method, then ClonableElement ensures that the hash value can only be computed on an immutable object. It furthermore caches it so that it is only computed once.

> **Warning:** for the hash caching mechanism to work correctly, the hash value cannot be 0.

EXAMPLES:

The following code shows a minimal example of usage of ClonableElement. We implement a class or pairs (x, y) such that $x < y$:

```
sage: from sage.structure.list_clone import ClonableElement
sage: class IntPair(ClonableElement):
....:      def __init__(self, parent, x, y):
....:          ClonableElement.__init__(self, parent=parent)
....:          self._x = x
....:          self._y = y
....:          self.set_immutable()
....:          self.check()
....:      def _repr_(self):
....:          return "(x=%s, y=%s)"%(self._x, self._y)
....:      def check(self):
....:          if self._x >= self._y:
....:              raise ValueError, "Incorrectly ordered pair"
....:      def get_x(self): return self._x
....:      def get_y(self): return self._y
....:      def set_x(self, v): self._require_mutable(); self._x = v
....:      def set_y(self, v): self._require_mutable(); self._y = v
```

Note: we don't need to define __copy__ since it is properly inherited from Element.

We now demonstrate the behavior. Let's create an IntPair:

```
sage: myParent = Parent()
sage: el = IntPair(myParent, 1, 3); el
(x=1, y=3)
sage: el.get_x()
1
```

Modifying it is forbidden:

```
sage: el.set_x(4)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

However, you can modify a mutable copy:

```
sage: with el.clone() as el1:
....:      el1.set_x(2)
sage: [el, el1]
[(x=1, y=3), (x=2, y=3)]
```

We check that the original and the modified copy are in a proper immutable state:

```
sage: el.is_immutable(), el1.is_immutable()
(True, True)
sage: el1.set_x(4)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

A modification which doesn't restore the invariant $x < y$ at the end is illegal and raise an exception:

```
sage: with el.clone() as elc2:
....:     elc2.set_x(5)
Traceback (most recent call last):
...
ValueError: Incorrectly ordered pair
```

clone(*check=True*)

Returns a clone that is mutable copy of `self`.

INPUT:

- `check` – a boolean indicating if `self.check()` must be called after modifications.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: el = IncreasingArrays()([1,2,3])
sage: with el.clone() as el1:
....:     el1[2] = 5
sage: el1
[1, 2, 5]
```

is_immutable()

Returns `True` if `self` is immutable (can not be changed) and `False` if it is not.

To make `self` immutable use `self.set_immutable()`.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: el = IncreasingArrays()([1,2,3])
sage: el.is_immutable()
True
sage: copy(el).is_immutable()
False
sage: with el.clone() as el1:
....:     print [el.is_immutable(), el1.is_immutable()]
[True, False]
```

is_mutable()

Returns `True` if `self` is mutable (can be changed) and `False` if it is not.

To make this object immutable use `self.set_immutable()`.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: el = IncreasingArrays()([1,2,3])
sage: el.is_mutable()
False
sage: copy(el).is_mutable()
True
sage: with el.clone() as el1:
```

```
....:      print [el.is_mutable(), el1.is_mutable()]
[False, True]
```

set_immutable()

Makes `self` immutable, so it can never again be changed.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: el = IncreasingArrays()([1,2,3])
sage: el1 = copy(el); el1.is_mutable()
True
sage: el1.set_immutable();  el1.is_mutable()
False
sage: el1[2] = 4
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

class sage.structure.list_clone.**ClonableIntArray**

Bases: `sage.structure.list_clone.ClonableElement`

Array of int with clone protocol

The class of objects which are `Element` behave as list of int and implement the clone protocol. See `ClonableElement` for details about clone protocol.

INPUT:

- `parent` – a `Parent`
- `lst` – a list
- `check` – a boolean specifying if the invariant must be checked using method `check()`
- `immutable` – a boolean telling wether the created element is immutable (defaults to `True`)

See also:

`IncreasingIntArray` for an example of usage.

check()

Check that `self` fulfill the invariants

This is an abstract method. Subclasses are supposed to overload `check`.

EXAMPLES:

```
sage: from sage.structure.list_clone import ClonableArray
sage: ClonableArray(Parent(), [1,2,3]) # indirect doctest
Traceback (most recent call last):
...
NotImplementedError: this should never be called, please overload the check method
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: el = IncreasingIntArrays()([1,2,4]) # indirect doctest
```

index(*item*)

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: c = IncreasingIntArrays()([1,2,4])
sage: c.index(1)
0
```

```
sage: c.index(4)
2
sage: c.index(5)
Traceback (most recent call last):
...
ValueError: list.index(x): x not in list
```

list()

Convert self into a Python list.

EXAMPLE:

```
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: I = IncreasingIntArrays()(range(5))
sage: I == range(5)
False
sage: I.list() == range(5)
True
sage: I = IncreasingIntArrays()(range(1000))
sage: I.list() == range(1000)
True
```

class sage.structure.list_clone.**ClonableList**

Bases: sage.structure.list_clone.ClonableArray

List with clone protocol

The class of objects which are Element behave as lists and implement the clone protocol. See ClonableElement for details about clone protocol.

See also:

IncreasingList for an example of usage.

append(*el*)

Appends el to self

INPUT: el – any object

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingLists
sage: el = IncreasingLists()([1])
sage: el.append(3)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with el.clone() as elc:
....:     elc.append(4)
....:     elc.append(6)
sage: elc
[1, 4, 6]
sage: with el.clone() as elc:
....:     elc.append(4)
....:     elc.append(2)
Traceback (most recent call last):
...
ValueError: array is not increasing
```

extend(*it*)

Extends self by the content of the iterable it

INPUT: `it` – any iterable

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingLists
sage: el = IncreasingLists()([1, 4, 5, 8, 9])
sage: el.extend((10,11))
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.

sage: with el.clone() as elc:
....:       elc.extend((10,11))
sage: elc
[1, 4, 5, 8, 9, 10, 11]

sage: el2 = IncreasingLists()([15, 16])
sage: with el.clone() as elc:
....:       elc.extend(el2)
sage: elc
[1, 4, 5, 8, 9, 15, 16]

sage: with el.clone() as elc:
....:       elc.extend((6,7))
Traceback (most recent call last):
...
ValueError: array is not increasing
```

insert (*index*, *el*)

Inserts `el` in `self` at position `index`

INPUT:

- `el` – any object
- `index` – any int

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingLists
sage: el = IncreasingLists()([1, 4, 5, 8, 9])
sage: el.insert(3, 6)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with el.clone() as elc:
....:       elc.insert(3, 6)
sage: elc
[1, 4, 5, 6, 8, 9]
sage: with el.clone() as elc:
....:       elc.insert(2, 6)
Traceback (most recent call last):
...
ValueError: array is not increasing
```

pop (*index=-1*)

Remove `self[index]` from `self` and returns it

INPUT: `index` - any int, default to -1

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingLists
sage: el = IncreasingLists()([1, 4, 5, 8, 9])
sage: el.pop()
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with el.clone() as elc:
....:     print elc.pop()
9
sage: elc
[1, 4, 5, 8]
sage: with el.clone() as elc:
....:     print elc.pop(2)
5
sage: elc
[1, 4, 8, 9]
```

remove (*el*)

Remove the first occurence of `el` from `self`

INPUT: `el` - any object

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingLists
sage: el = IncreasingLists()([1, 4, 5, 8, 9])
sage: el.remove(4)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with el.clone() as elc:
....:     elc.remove(4)
sage: elc
[1, 5, 8, 9]
sage: with el.clone() as elc:
....:     elc.remove(10)
Traceback (most recent call last):
...
ValueError: list.remove(x): x not in list
```

class `sage.structure.list_clone.`**NormalizedClonableList**

Bases: `sage.structure.list_clone.ClonableList`

List with clone protocol and normal form

This is a subclass of `ClonableList` which call a method `normalize()` at creation and after any modification of its instance.

See also:

`SortedList` for an example of usage.

EXAMPLES:

We construct a sorted list through its parent:

```
sage: from sage.structure.list_clone_demo import SortedLists
sage: SL = SortedLists()
sage: sl1 = SL([4,2,6,1]); sl1
[1, 2, 4, 6]
```

Normalization is also performed atfer modification:

```
sage: with sl1.clone() as sl2:
....:       sl2[1] = 12
sage: sl2
[1, 4, 6, 12]
```

normalize()

Normalize `self`

This is an abstract method. Subclasses are supposed to overload `normalize()`. The call `self.normalize()` is supposed to

- call `self._require_mutable()` to check that `self` is in a proper mutable state
- modify `self` to put it in a normal form

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import SortedList, SortedLists
sage: l = SortedList(SortedLists(), [2,3,2], False, False)
sage: l
[2, 2, 3]
sage: l.check()
Traceback (most recent call last):
...
ValueError: list is not strictly increasing
```

CHAPTER

FOURTEEN

ELEMENTS, ARRAY AND LISTS WITH CLONE PROTOCOL, DEMONSTRATION CLASSES

This module demonstrate the usage of the various classes defined in `list_clone`

class `sage.structure.list_clone_demo.`**`IncreasingArray`**

Bases: `sage.structure.list_clone.ClonableArray`

A small extension class for testing `ClonableArray`.

TESTS:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: TestSuite(IncreasingArrays()([1,2,3])).run()
sage: TestSuite(IncreasingArrays()([])).run()
```

`check`()

Check that `self` is increasing.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: IncreasingArrays()([1,2,3]) # indirect doctest
[1, 2, 3]
sage: IncreasingArrays()([3,2,1]) # indirect doctest
Traceback (most recent call last):
...
ValueError: array is not increasing
```

class `sage.structure.list_clone_demo.`**`IncreasingArrays`**

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.Parent`

A small (incomplete) parent for testing `ClonableArray`

TESTS:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: IncreasingArrays().element_class
<type 'sage.structure.list_clone_demo.IncreasingArray'>
```

`Element`

alias of `IncreasingArray`

class `sage.structure.list_clone_demo.`**`IncreasingIntArray`**

Bases: `sage.structure.list_clone.ClonableIntArray`

A small extension class for testing `ClonableIntArray`.

TESTS:

```
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: TestSuite(IncreasingIntArrays()([1,2,3])).run()
sage: TestSuite(IncreasingIntArrays()([])).run()
```

check()

Check that `self` is increasing.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: IncreasingIntArrays()([1,2,3]) # indirect doctest
[1, 2, 3]
sage: IncreasingIntArrays()([3,2,1]) # indirect doctest
Traceback (most recent call last):
...
ValueError: array is not increasing
```

class sage.structure.list_clone_demo.**IncreasingIntArrays**

Bases: `sage.structure.list_clone_demo.IncreasingArrays`

A small (incomplete) parent for testing `ClonableIntArray`

TESTS:

```
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: IncreasingIntArrays().element_class
<type 'sage.structure.list_clone_demo.IncreasingIntArray'>
```

Element

alias of `IncreasingIntArray`

class sage.structure.list_clone_demo.**IncreasingList**

Bases: `sage.structure.list_clone.ClonableList`

A small extension class for testing `ClonableList`

TESTS:

```
sage: from sage.structure.list_clone_demo import IncreasingLists
sage: TestSuite(IncreasingLists()([1,2,3])).run()
sage: TestSuite(IncreasingLists()([])).run()
```

check()

Check that `self` is increasing

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingLists
sage: IncreasingLists()([1,2,3]) # indirect doctest
[1, 2, 3]
sage: IncreasingLists()([3,2,1]) # indirect doctest
Traceback (most recent call last):
...
ValueError: array is not increasing
```

class sage.structure.list_clone_demo.**IncreasingLists**

Bases: `sage.structure.list_clone_demo.IncreasingArrays`

A small (incomplete) parent for testing `ClonableList`

TESTS:

```
sage: from sage.structure.list_clone_demo import IncreasingLists
sage: IncreasingLists().element_class
<type 'sage.structure.list_clone_demo.IncreasingList'>
```

Element
alias of `IncreasingList`

class `sage.structure.list_clone_demo.`**`SortedList`**
Bases: `sage.structure.list_clone.NormalizedClonableList`

A small extension class for testing `NormalizedClonableList`.

TESTS:

```
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: TestSuite(IncreasingIntArrays()([1,2,3])).run()
sage: TestSuite(IncreasingIntArrays()([])).run()
```

check()
Check that `self` is strictly increasing

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import SortedList, SortedLists
sage: SortedLists()([1,2,3]) # indirect doctest
[1, 2, 3]
sage: SortedLists()([3,2,2]) # indirect doctest
Traceback (most recent call last):
...
ValueError: list is not strictly increasing
```

normalize()
Normalize `self`

Sort the list stored in `self`.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import SortedList, SortedLists
sage: l = SortedList(SortedLists(), [3,1,2], False, False)
sage: l          # indirect doctest
[1, 2, 3]
sage: l[1] = 5; l
[1, 5, 3]
sage: l.normalize(); l
[1, 3, 5]
```

class `sage.structure.list_clone_demo.`**`SortedLists`**
Bases: `sage.structure.list_clone_demo.IncreasingLists`

A small (incomplete) parent for testing `NormalizedClonableList`

TESTS:

```
sage: from sage.structure.list_clone_demo import SortedList, SortedLists
sage: SL = SortedLists()
sage: SL([3,1,2])
[1, 2, 3]
```

Element
alias of `SortedList`

CHAPTER

FIFTEEN

MUTABILITY CYTHON IMPLEMENTATION

class sage.structure.mutability.**Mutability**

Bases: object

is_immutable()

Return True if this object is immutable (can not be changed) and False if it is not.

To make this object immutable use self.set_immutable().

EXAMPLE:

```
sage: v = Sequence([1,2,3,4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.is_immutable()
False
sage: v.set_immutable()
sage: v.is_immutable()
True
```

is_mutable()

set_immutable()

Make this object immutable, so it can never again be changed.

EXAMPLES:

```
sage: v = Sequence([1,2,3,4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.set_immutable()
sage: v[3] = 7
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

sage.structure.mutability.**require_immutable**(f)

A decorator that requires mutability for a method to be called.

EXAMPLES:

```
sage: from sage.structure.mutability import require_mutable, require_immutable
sage: class A:
...     def __init__(self, val):
...         self._m = val
...     @require_mutable
...     def change(self, new_val):
```

```
...         'change self'
...         self._m = new_val
...     @require_immutable
...     def __hash__(self):
...         'implement hash'
...         return hash(self._m)
sage: a = A(5)
sage: a.change(6)
sage: hash(a)    # indirect doctest
Traceback (most recent call last):
...
ValueError: <type 'instance'> instance is mutable, <function __hash__ at ...> must not be called
sage: a._is_immutable = True
sage: hash(a)
6
sage: a.change(7)
Traceback (most recent call last):
...
ValueError: <type 'instance'> instance is immutable, <function change at ...> must not be called
sage: from sage.misc.sageinspect import sage_getdoc
sage: print sage_getdoc(a.__hash__)
implement hash
```

AUTHORS:

- Simon King <simon.king@uni-jena.de>

sage.structure.mutability.**require_mutable**(*f*)

A decorator that requires mutability for a method to be called.

EXAMPLES:

```
sage: from sage.structure.mutability import require_mutable, require_immutable
sage: class A:
...     def __init__(self, val):
...         self._m = val
...     @require_mutable
...     def change(self, new_val):
...         'change self'
...         self._m = new_val
...     @require_immutable
...     def __hash__(self):
...         'implement hash'
...         return hash(self._m)
sage: a = A(5)
sage: a.change(6)
sage: hash(a)
Traceback (most recent call last):
...
ValueError: <type 'instance'> instance is mutable, <function __hash__ at ...> must not be called
sage: a._is_immutable = True
sage: hash(a)
6
sage: a.change(7)    # indirect doctest
Traceback (most recent call last):
...
ValueError: <type 'instance'> instance is immutable, <function change at ...> must not be called
sage: from sage.misc.sageinspect import sage_getdoc
sage: print sage_getdoc(a.change)
```

`change self`

AUTHORS:

- Simon King <simon.king@uni-jena.de>

CHAPTER

SIXTEEN

SEQUENCES

A mutable sequence of elements with a common guaranteed category, which can be set immutable.

Sequence derives from list, so has all the functionality of lists and can be used wherever lists are used. When a sequence is created without explicitly given the common universe of the elements, the constructor coerces the first and second element to some *canonical* common parent, if possible, then the second and third, etc. If this is possible, it then coerces everything into the canonical parent at the end. (Note that canonical coercion is very restrictive.) The sequence then has a function `universe()` which returns either the common canonical parent (if the coercion succeeded), or the category of all objects (Objects()). So if you have a list v and type

> sage: v = [1, 2/3, 5] sage: w = Sequence(v) sage: w.universe() Rational Field

then since `w.universe()` is $\mathbf{Q}$, you're guaranteed that all elements of w are rationals:

```
sage: v[0].parent()
Integer Ring
sage: w[0].parent()
Rational Field
```

If you do assignment to w this property of being rationals is guaranteed to be preserved.

> sage: w[0] = 2 sage: w[0].parent() Rational Field sage: w[0] = 'hi' Traceback (most recent call last): ... TypeError: unable to convert hi to a rational

However, if you do `w = Sequence(v)` and the resulting universe is `Objects()`, the elements are not guaranteed to have any special parent. This is what should happen, e.g., with finite field elements of different characteristics:

```
sage: v = Sequence([GF(3)(1), GF(7)(1)])
sage: v.universe()
Category of objects
```

You can make a list immutable with `v.freeze()`. Assignment is never again allowed on an immutable list.

Creation of a sequence involves making a copy of the input list, and substantial coercions. It can be greatly sped up by explicitly specifying the universe of the sequence:

```
sage: v = Sequence(range(10000), universe=ZZ)
```

TESTS:

```
sage: v = Sequence([1..5])
sage: loads(dumps(v)) == v
True
```

`sage.structure.sequence.`**`Sequence`**(*x*, *universe=None*, *check=True*, *immutable=False*, *cr=False*, *cr_str=None*, *use_sage_types=False*)

A mutable list of elements with a common guaranteed universe, which can be set immutable.

A universe is either an object that supports coercion (e.g., a parent), or a category.

INPUT:

- x - a list or tuple instance
- universe - (default: None) the universe of elements; if None determined using canonical coercions and the entire list of elements. If list is empty, is category Objects() of all objects.
- check – (default: True) whether to coerce the elements of x into the universe
- immutable - (default: True) whether or not this sequence is immutable
- cr - (default: False) if True, then print a carriage return after each comma when printing this sequence.
- cr_str - (default: False) if True, then print a carriage return after each comma when calling str() on this sequence.
- **use_sage_types – (default: False) if True, coerce the** built-in Python numerical types int, long, float, complex to the corresponding Sage types (this makes functions like vector() more flexible)

OUTPUT:

- a sequence

EXAMPLES:

```
sage: v = Sequence(range(10))
sage: v.universe()
<type 'int'>
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We can request that the built-in Python numerical types be coerced to Sage objects:

```
sage: v = Sequence(range(10), use_sage_types=True)
sage: v.universe()
Integer Ring
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You can also use seq for “Sequence”, which is identical to using Sequence:

```
sage: v = seq([1,2,1/1]); v
[1, 2, 1]
sage: v.universe()
Rational Field
sage: v.parent()
Category of sequences in Rational Field
sage: v.parent()([3,4/3])
[3, 4/3]
```

Note that assignment coerces if possible,:

```
sage: v = Sequence(range(10), ZZ)
sage: a = QQ(5)
sage: v[3] = a
sage: parent(v[3])
Integer Ring
sage: parent(a)
Rational Field
sage: v[3] = 2/3
Traceback (most recent call last):
```

```
...
TypeError: no conversion of this rational to integer
```

Sequences can be used absolutely anywhere lists or tuples can be used:

```
sage: isinstance(v, list)
True
```

Sequence can be immutable, so entries can't be changed:

```
sage: v = Sequence([1,2,3], immutable=True)
sage: v.is_immutable()
True
sage: v[0] = 5
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

Only immutable sequences are hashable (unlike Python lists), though the hashing is potentially slow, since it first involves conversion of the sequence to a tuple, and returning the hash of that.:

```
sage: v = Sequence(range(10), ZZ, immutable=True)
sage: hash(v)
1591723448             # 32-bit
-4181190870548101704   # 64-bit
```

If you really know what you are doing, you can circumvent the type checking (for an efficiency gain):

```
sage: list.__setitem__(v, int(1), 2/3)        # bad circumvention
sage: v
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
sage: list.__setitem__(v, int(1), int(2))     # not so bad circumvention
```

You can make a sequence with a new universe from an old sequence.:

```
sage: w = Sequence(v, QQ)
sage: w
[0, 2, 2, 3, 4, 5, 6, 7, 8, 9]
sage: w.universe()
Rational Field
sage: w[1] = 2/3
sage: w
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
```

Sequences themselves live in a category, the category of all sequences in the given universe.:

```
sage: w.category()
Category of sequences in Rational Field
```

This is also the parent of any sequence:

```
sage: w.parent()
Category of sequences in Rational Field
```

The default universe for any sequence, if no compatible parent structure can be found, is the universe of all Sage objects.

This example illustrates how every element of a list is taken into account when constructing a sequence.:

```
sage: v = Sequence([1,7,6,GF(5)(3)]); v
[1, 2, 1, 3]
sage: v.universe()
```

```
Finite Field of size 5
sage: v.parent()
Category of sequences in Finite Field of size 5
sage: v.parent()([7,8,9])
[2, 3, 4]
```

class `sage.structure.sequence.`**`Sequence_generic`**(*x, universe=None, check=True, immutable=False, cr=False, cr_str=None, use_sage_types=False*)

Bases: `sage.structure.sage_object.SageObject`, `list`

A mutable list of elements with a common guaranteed universe, which can be set immutable.

A universe is either an object that supports coercion (e.g., a parent), or a category.

INPUT:

- `x` - a list or tuple instance
- `universe` - (default: None) the universe of elements; if None determined using canonical coercions and the entire list of elements. If list is empty, is category Objects() of all objects.
- `check` – (default: True) whether to coerce the elements of x into the universe
- `immutable` - (default: True) whether or not this sequence is immutable
- `cr` - (default: False) if True, then print a carriage return after each comma when printing this sequence.
- **`use_sage_types` – (default: False) if True, coerce the** built-in Python numerical types int, long, float, complex to the corresponding Sage types (this makes functions like vector() more flexible)

OUTPUT:

- a sequence

EXAMPLES:

```
sage: v = Sequence(range(10))
sage: v.universe()
<type 'int'>
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We can request that the built-in Python numerical types be coerced to Sage objects:

```
sage: v = Sequence(range(10), use_sage_types=True)
sage: v.universe()
Integer Ring
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You can also use seq for “Sequence”, which is identical to using Sequence:

```
sage: v = seq([1,2,1/1]); v
[1, 2, 1]
sage: v.universe()
Rational Field
sage: v.parent()
Category of sequences in Rational Field
sage: v.parent()([3,4/3])
[3, 4/3]
```

Note that assignment coerces if possible,

```
sage: v = Sequence(range(10), ZZ)
sage: a = QQ(5)
sage: v[3] = a
sage: parent(v[3])
Integer Ring
sage: parent(a)
Rational Field
sage: v[3] = 2/3
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

Sequences can be used absolutely anywhere lists or tuples can be used:

```
sage: isinstance(v, list)
True
```

Sequence can be immutable, so entries can't be changed:

```
sage: v = Sequence([1,2,3], immutable=True)
sage: v.is_immutable()
True
sage: v[0] = 5
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

Only immutable sequences are hashable (unlike Python lists), though the hashing is potentially slow, since it first involves conversion of the sequence to a tuple, and returning the hash of that.

```
sage: v = Sequence(range(10), ZZ, immutable=True)
sage: hash(v)
1591723448             # 32-bit
-4181190870548101704   # 64-bit
```

If you really know what you are doing, you can circumvent the type checking (for an efficiency gain):

```
sage: list.__setitem__(v, int(1), 2/3)        # bad circumvention
sage: v
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
sage: list.__setitem__(v, int(1), int(2))     # not so bad circumvention
```

You can make a sequence with a new universe from an old sequence.

```
sage: w = Sequence(v, QQ)
sage: w
[0, 2, 2, 3, 4, 5, 6, 7, 8, 9]
sage: w.universe()
Rational Field
sage: w[1] = 2/3
sage: w
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
```

Sequences themselves live in a category, the category of all sequences in the given universe.

```
sage: w.category()
Category of sequences in Rational Field
```

This is also the parent of any sequence:

```
sage: w.parent()
Category of sequences in Rational Field
```

The default universe for any sequence, if no compatible parent structure can be found, is the universe of all Sage objects.

This example illustrates how every element of a list is taken into account when constructing a sequence.

```
sage: v = Sequence([1,7,6,GF(5)(3)]); v
[1, 2, 1, 3]
sage: v.universe()
Finite Field of size 5
sage: v.parent()
Category of sequences in Finite Field of size 5
sage: v.parent()([7,8,9])
[2, 3, 4]
```

append(*x*)

EXAMPLES: sage: v = Sequence([1,2,3,4], immutable=True) sage: v.append(34) Traceback (most recent call last): ... ValueError: object is immutable; please change a copy instead. sage: v = Sequence([1/3,2,3,4]) sage: v.append(4) sage: type(v[4]) <type 'sage.rings.rational.Rational'>

category()

EXAMPLES:

```
sage: Sequence([1,2/3,-2/5]).category()
Category of sequences in Rational Field
```

extend(*iterable*)

Extend list by appending elements from the iterable.

EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.extend(range(4))
sage: B
[1, 2, 3, 0, 1, 2, 3]
```

insert(*index*, *object*)

Insert object before index.

EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.insert(10, 5)
sage: B
[1, 2, 3, 5]
```

is_immutable()

Return True if this object is immutable (can not be changed) and False if it is not.

To make this object immutable use `set_immutable()`.

EXAMPLE:

```
sage: v = Sequence([1,2,3,4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.is_immutable()
False
```

```
sage: v.set_immutable()
sage: v.is_immutable()
True
```

is_mutable()

EXAMPLES:

```
sage: a = Sequence([1,2/3,-2/5])
sage: a.is_mutable()
True
sage: a[0] = 100
sage: type(a[0])
<type 'sage.rings.rational.Rational'>
sage: a.set_immutable()
sage: a[0] = 50
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: a.is_mutable()
False
```

parent()

EXAMPLES:

```
sage: Sequence([1,2/3,-2/5]).parent()
Category of sequences in Rational Field
```

pop(*index=-1*)

Remove and return item at index (default last)

EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.pop(1)
2
sage: B
[1, 3]
```

remove(*value*)

Remove first occurrence of value

EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.remove(2)
sage: B
[1, 3]
```

reverse()

Reverse the elements of self, in place.

EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.reverse(); B
[3, 2, 1]
```

set_immutable()

Make this object immutable, so it can never again be changed.

EXAMPLES:

```
sage: v = Sequence([1,2,3,4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.set_immutable()
sage: v[3] = 7
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

sort (*cmp=None*, *key=None*, *reverse=False*)

Sort this list *IN PLACE*.

cmp(x, y) -> -1, 0, 1

EXAMPLES:

```
sage: B = Sequence([3,2,1/5])
sage: B.sort()
sage: B
[1/5, 2, 3]
sage: B.sort(reverse=True); B
[3, 2, 1/5]
sage: B.sort(cmp = lambda x,y: cmp(y,x)); B
[3, 2, 1/5]
sage: B.sort(cmp = lambda x,y: cmp(y,x), reverse=True); B
[1/5, 2, 3]
```

universe ()

EXAMPLES:

```
sage: Sequence([1,2/3,-2/5]).universe()
Rational Field
sage: Sequence([1,2/3,'-2/5']).universe()
Category of objects
```

sage.structure.sequence.**seq** (*x*, *universe=None*, *check=True*, *immutable=False*, *cr=False*, *cr_str=None*, *use_sage_types=False*)

A mutable list of elements with a common guaranteed universe, which can be set immutable.

A universe is either an object that supports coercion (e.g., a parent), or a category.

INPUT:

- `x` - a list or tuple instance
- `universe` - (default: None) the universe of elements; if None determined using canonical coercions and the entire list of elements. If list is empty, is category Objects() of all objects.
- `check` – (default: True) whether to coerce the elements of x into the universe
- `immutable` - (default: True) whether or not this sequence is immutable
- `cr` - (default: False) if True, then print a carriage return after each comma when printing this sequence.
- `cr_str` - (default: False) if True, then print a carriage return after each comma when calling `str()` on this sequence.
- **`use_sage_types` – (default: False) if True, coerce the** built-in Python numerical types int, long, float, complex to the corresponding Sage types (this makes functions like vector() more flexible)

OUTPUT:

- a sequence

EXAMPLES:

```
sage: v = Sequence(range(10))
sage: v.universe()
<type 'int'>
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We can request that the built-in Python numerical types be coerced to Sage objects:

```
sage: v = Sequence(range(10), use_sage_types=True)
sage: v.universe()
Integer Ring
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You can also use seq for "Sequence", which is identical to using Sequence:

```
sage: v = seq([1,2,1/1]); v
[1, 2, 1]
sage: v.universe()
Rational Field
sage: v.parent()
Category of sequences in Rational Field
sage: v.parent()([3,4/3])
[3, 4/3]
```

Note that assignment coerces if possible,:

```
sage: v = Sequence(range(10), ZZ)
sage: a = QQ(5)
sage: v[3] = a
sage: parent(v[3])
Integer Ring
sage: parent(a)
Rational Field
sage: v[3] = 2/3
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

Sequences can be used absolutely anywhere lists or tuples can be used:

```
sage: isinstance(v, list)
True
```

Sequence can be immutable, so entries can't be changed:

```
sage: v = Sequence([1,2,3], immutable=True)
sage: v.is_immutable()
True
sage: v[0] = 5
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

Only immutable sequences are hashable (unlike Python lists), though the hashing is potentially slow, since it first involves conversion of the sequence to a tuple, and returning the hash of that.:

```
sage: v = Sequence(range(10), ZZ, immutable=True)
sage: hash(v)
1591723448            # 32-bit
-4181190870548101704  # 64-bit
```

If you really know what you are doing, you can circumvent the type checking (for an efficiency gain):

```
sage: list.__setitem__(v, int(1), 2/3)        # bad circumvention
sage: v
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
sage: list.__setitem__(v, int(1), int(2))     # not so bad circumvention
```

You can make a sequence with a new universe from an old sequence.:

```
sage: w = Sequence(v, QQ)
sage: w
[0, 2, 2, 3, 4, 5, 6, 7, 8, 9]
sage: w.universe()
Rational Field
sage: w[1] = 2/3
sage: w
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
```

Sequences themselves live in a category, the category of all sequences in the given universe.:

```
sage: w.category()
Category of sequences in Rational Field
```

This is also the parent of any sequence:

```
sage: w.parent()
Category of sequences in Rational Field
```

The default universe for any sequence, if no compatible parent structure can be found, is the universe of all Sage objects.

This example illustrates how every element of a list is taken into account when constructing a sequence.:

```
sage: v = Sequence([1,7,6,GF(5)(3)]); v
[1, 2, 1, 3]
sage: v.universe()
Finite Field of size 5
sage: v.parent()
Category of sequences in Finite Field of size 5
sage: v.parent()([7,8,9])
[2, 3, 4]
```

CHAPTER

SEVENTEEN

ELEMENT WRAPPER

Wrapping Sage or Python objects as Sage elements.

AUTHORS:

- Nicolas Thiery (2008-2010): Initial version
- Travis Scrimshaw (2013-05-04): Cythonized version

class sage.structure.element_wrapper.**DummyParent**(*name*)

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

A class for creating dummy parents for testing ElementWrapper

class sage.structure.element_wrapper.**ElementWrapper**

Bases: sage.structure.element.Element

A class for wrapping Sage or Python objects as Sage elements.

EXAMPLES:

```
sage: from sage.structure.element_wrapper import DummyParent
sage: parent = DummyParent("A parent")
sage: o = ElementWrapper(parent, "bla"); o
'bla'
sage: isinstance(o, sage.structure.element.Element)
True
sage: o.parent()
A parent
sage: o.value
'bla'
```

Note that o is not *an instance of* str, but rather *contains a* str. Therefore, o does not inherit the string methods. On the other hand, it is provided with reasonable default implementations for equality testing, hashing, etc.

The typical use case of ElementWrapper is for trivially constructing new element classes from pre-existing Sage or Python classes, with a containment relation. Here we construct the tropical monoid of integers endowed with min as multiplication. There, it is desirable *not* to inherit the factor method from Integer:

```
sage: class MinMonoid(Parent):
....:     def _repr_(self):
....:         return "The min monoid"
....:
sage: M = MinMonoid()
sage: class MinMonoidElement(ElementWrapper):
....:     wrapped_class = Integer
....:
```

```
....:     def __mul__(self, other):
....:         return MinMonoidElement(self.parent(), min(self.value, other.value))
sage: x = MinMonoidElement(M, 5); x
5
sage: x.parent()
The min monoid
sage: x.value
5
sage: y = MinMonoidElement(M, 3)
sage: x * y
3
```

This example was voluntarily kept to a bare minimum. See the examples in the categories (e.g. `Semigroups().example()`) for several full featured applications.

> **Warning:** Versions before trac ticket #14519 had parent as the second argument and the value as the first.

value

class sage.structure.element_wrapper.**ElementWrapperTester**

Bases: `sage.structure.element_wrapper.ElementWrapper`

Test class for the default `__copy__()` method of subclasses of `ElementWrapper`.

TESTS:

```
sage: from sage.structure.element_wrapper import ElementWrapperTester
sage: x = ElementWrapperTester()
sage: x.append(2); y = copy(x); y.append(42)
sage: type(y)
<class 'sage.structure.element_wrapper.ElementWrapperTester'>
sage: x, y
([n=1, value=[2]], [n=2, value=[2, 42]])
sage: x.append(21); x.append(7)
sage: x, y
([n=3, value=[2, 21, 7]], [n=2, value=[2, 42]])
sage: x.value, y.value
([2, 21, 7], [2, 42])
sage: x.__dict__, y.__dict__
({'n': 3}, {'n': 2})
```

append(*x*)

TESTS:

```
sage: from sage.structure.element_wrapper import ElementWrapperTester
sage: x = ElementWrapperTester()
sage: x.append(2); x
[n=1, value=[2]]
```

CHAPTER

EIGHTEEN

INDEXED GENERATORS

class sage.structure.indexed_generators.**IndexedGenerators**(*indices*, *prefix='x'*, ***kwds*)

Bases: object

Abstract base class for parents whose elements consist of generators indexed by an arbitrary set.

Options controlling the printing of elements:

- prefix – string, prefix used for printing elements of this module (optional, default 'x'). With the default, a monomial indexed by 'a' would be printed as x['a'].
- latex_prefix – string or None, prefix used in the LaTeX representation of elements (optional, default None). If this is anything except the empty string, it prints the index as a subscript. If this is None, it uses the setting for prefix, so if prefix is set to "B", then a monomial indexed by 'a' would be printed as B_{a}. If this is the empty string, then don't print monomials as subscripts: the monomial indexed by 'a' would be printed as a, or as [a] if latex_bracket is True.
- bracket – None, bool, string, or list or tuple of strings (optional, default None): if None, use the value of the attribute self._repr_option_bracket, which has default value True. (self._repr_option_bracket is available for backwards compatibility. Users should set bracket instead. If bracket is set to anything except None, it overrides the value of self._repr_option_bracket.) If False, do not include brackets when printing elements: a monomial indexed by 'a' would be printed as B'a', and a monomial indexed by (1,2,3) would be printed as B(1,2,3). If True, use "[" and "]" as brackets. If it is one of "[", "(", or "{", use it and its partner as brackets. If it is any other string, use it as both brackets. If it is a list or tuple of strings, use the first entry as the left bracket and the second entry as the right bracket.
- latex_bracket – bool, string, or list or tuple of strings (optional, default False): if False, do not include brackets in the LaTeX representation of elements. This option is only relevant if latex_prefix is the empty string; otherwise, brackets are not used regardless. If True, use "left[" and "right]" as brackets. If this is one of "[", "(", "\{", "|", or "||", use it and its partner, prepended with "left" and "right", as brackets. If this is any other string, use it as both brackets. If this is a list or tuple of strings, use the first entry as the left bracket and the second entry as the right bracket.
- scalar_mult – string to use for scalar multiplication in the print representation (optional, default "*")
- latex_scalar_mult – string or None (default: None), string to use for scalar multiplication in the latex representation. If None, use the empty string if scalar_mult is set to "*", otherwise use the value of scalar_mult.
- tensor_symbol – string or None (default: None), string to use for tensor product in the print representation. If None, use sage.categories.tensor.symbol.
- generator_cmp – a comparison function (default: cmp), to use for sorting elements in the output of elements

Note: These print options may also be accessed and modified using the `print_options()` method, after the parent has been defined.

EXAMPLES:

We demonstrate a variety of the input options:

```
sage: from sage.structure.indexed_generators import IndexedGenerators
sage: I = IndexedGenerators(ZZ, prefix='A')
sage: I._repr_generator(2)
'A[2]'
sage: I._latex_generator(2)
'A_{2}'

sage: I = IndexedGenerators(ZZ, bracket='(')
sage: I._repr_generator(2)
'x(2)'
sage: I._latex_generator(2)
'x_{2}'

sage: I = IndexedGenerators(ZZ, prefix="", latex_bracket='(')
sage: I._repr_generator(2)
'[2]'
sage: I._latex_generator(2)
\left( 2 \right)

sage: I = IndexedGenerators(ZZ, bracket=['|', '>'])
sage: I._repr_generator(2)
'x|2>'
```

indices()

Return the indices of `self`.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.indices()
{'a', 'b', 'c'}
```

prefix()

Return the prefix used when displaying elements of self.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.prefix()
'B'

sage: X = SchubertPolynomialRing(QQ)
sage: X.prefix()
'X'
```

print_options(***kwds*)

Return the current print options, or set an option.

INPUT: all of the input is optional; if present, it should be in the form of keyword pairs, such as `latex_bracket='('`. The allowable keywords are:

- `prefix`

- latex_prefix
- bracket
- latex_bracket
- scalar_mult
- latex_scalar_mult
- tensor_symbol
- generator_cmp

See the documentation for CombinatorialFreeModule for descriptions of the effects of setting each of these options.

OUTPUT: if the user provides any input, set the appropriate option(s) and return nothing. Otherwise, return the dictionary of settings for print and LaTeX representations.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, [1,2,3], prefix='x')
sage: F.print_options()
{...'prefix': 'x'...}
sage: F.print_options(bracket='(')
sage: F.print_options()
{...'bracket': '('...}
```

TESTS:

```
sage: sorted(F.print_options().items())
[('bracket', '('), ('generator_cmp', <built-in function cmp>),
 ('latex_bracket', False), ('latex_prefix', None),
 ('latex_scalar_mult', None), ('prefix', 'x'),
 ('scalar_mult', '*'), ('tensor_symbol', None)]
sage: F.print_options(bracket='[') # reset
```

CHAPTER

NINETEEN

GLOBAL OPTIONS

The GlobalOptions class provides a generic mechanism for setting and accessing **global** options for parents in one or several related classes, typically for customizing the representation of their elements. This class will eventually also support setting options on a parent by parent basis.

See also:

For better examples of GlobalOptions in action see sage.combinat.partition.Partitions.global_options() and sage.combinat.tableau.Tableaux.global_options().

19.1 Construction of options classes

The general setup for creating a set of global options is:

```
MyOptions=GlobalOptions('option name',
    doc='Nice options',
    first_option=dict(default='default value',
                      description='Changes the functionality of _repr_',
                      values=dict(with_bells='causes _repr_ to print with bells',
                                  with_whistles='causes _repr_ to print with whistles',
                                  ...),
                      aliases=dict(bells='option1', whistles='option2', ...),
    second_option=dict(...),
    third_option=dict(),
    end_doc='end of options documentation'
)
```

Each option is specified as a dictionary which describes the possible values for the option and its documentation. The possible entries in this dictionary are:

- alias – Allows for several option values to do the same thing.
- alt_name – An alternative name for this option.
- checker – A validation function which returns whether a user supplied value is valid or not. This is typically useful for large lists of legal values such as $\mathbf{N}$.
- default – Gives the default value for the option.
- description – A one line description of the option.
- link_to – Links this option to another one in another set of global options. This is used for example to allow Partitions and Tableaux to share the same convention option.
- setter – A function which is called **after** the value of the option is changed.
- values – A dictionary assigning each valid value for the option to a short description of what it does.

- case_sensitive – (Default: True) True or False depending on whether the values of the option are case sensitive.

For each option, either a complete list of possible values, via values, or a validation function, via checker, must be given. The values can be quite arbitrary, including user-defined functions which customize the default behaviour of the classes such as the output of _repr_ or latex(). See *Dispatchers* below, and dispatcher(), for more information.

The documentation for the options is automatically constructed by combining the description of each option with a header and footer which are given by the following optional, but recommended, arguments:

- doc – The top half of the documentation which appears before the automatically generated list of options and their possible values.
- end_doc – The second half of the documentation which appears after the list of options and their values.

The basic structure for defining a GlobalOptions class is best illustrated by an example:

```
sage: from sage.structure.global_options import GlobalOptions
sage: menu=GlobalOptions('menu', doc='Fancy documentation\n'+'-'*19, end_doc='The END!',
...       entree=dict(default='soup',
...                   description='The first course of a meal',
...                   values=dict(soup='soup of the day', bread='oven baked'),
...                   alias=dict(rye='bread')),
...       appetizer=dict(alt_name='entree'),
...       main=dict(default='pizza', description='Main meal',
...                 values=dict(pizza='thick crust', pasta='penne arrabiata'),
...                 case_sensitive=False),
...       dessert=dict(default='espresso', description='Dessert',
...                    values=dict(espresso='life begins again',
...                                cake='waist begins again',
...                                cream='fluffy, white stuff')),
...       tip=dict(default=10, description='Reward for good service',
...       checker=lambda tip: tip in range(0,20))
...   )
sage: menu
options for menu
```

For more details see GlobalOptions.

19.2 Accessing and setting option values

All options and their values, when they are strings, are forced to be lower case. The values of an options class can be set and accessed by calling the class or by treating the class as an array.

Continuing the example from *Construction of options classes*:

```
sage: menu()
Current options for menu
  - dessert: espresso
  - entree:  soup
  - main:    pizza
  - tip:     10
sage: menu('dessert')
'espresso'
sage: menu['dessert']
'espresso'
```

Note that, provided there is no ambiguity, options and their values can be abbreviated:

```
sage: menu['d']
'espresso'
sage: menu('m','t',des='esp', ent='sou')  # get and set several values at once
['pizza', 10]
sage: menu(t=15); menu['tip']
15
sage: menu(e='s', m='Pi'); menu()
Current options for menu
  - dessert: espresso
  - entree:  soup
  - main:    pizza
  - tip:     15
sage: menu(m='P')
Traceback (most recent call last):
...
ValueError: P is not a valid value for main in the options for menu
```

19.3 Setter functions

Each option of a `GlobalOptions` can be equipped with an optional setter function which is called **after** the value of the option is changed. In the following example, setting the option 'add' changes the state of the class by setting an attribute in this class using a `classmethod()`. Note that the options object is inserted after the creation of the class in order to access the `classmethod()` as `A.setter`:

```
sage: from sage.structure.global_options import GlobalOptions
sage: class A(SageObject):
...       state = 0
...       @classmethod
...       def setter(cls, option, val):
...           cls.state += int(val)
...
sage: A.options=GlobalOptions('A',
...                           add=dict(default=1,
...                                    checker=lambda v: int(v)>0,
...                                    description='An option with a setter',
...                                    setter=A.setter))
sage: a = A(2); a.state
1
sage: a.options()
Current options for A
- add: 1
sage: a.options(add=4)
sage: a.state
5
sage: a.options()
Current options for A
- add: 4
```

Another alternative is to construct the options class inside the `__init__` method of the class `A`.

19.4 Documentation for options

The documentation for a `GlobalOptions` is automatically generated from the supplied options. For example, the generated documentation for the options `menu` defined in *Construction of options classes* is the following:

```
Fancy documentation
-------------------

OPTIONS:

- ''appetizer'' -- alternative name for ''entree''

- ''dessert'' -- (default: ''espresso'')
  Dessert

  - ''cake''     -- waist begins again
  - ''cream''    -- fluffy, white stuff
  - ''espresso'' -- life begins again

- ''entree'' -- (default: ''soup'')
  The first course of a meal

  - ''bread'' -- oven baked
  - ''rye''   -- alias for bread
  - ''soup''  -- soup of the day

- ''main'' -- (default: ''pizza'')
  Main meal

  - ''pasta'' -- penne arrabiata
  - ''pizza'' -- thick crust

- tip -- (default: 10)
  Reward for good service

The END!

See :class:'~sage.structure.global_options.GlobalOptions' for more features of these options.
```

In addition, help on each option, and its list of possible values, can be obtained by (trying to) set the option equal to '?':

```
sage: menu(des='?')
- ''dessert'' -- (default: ''espresso'')
  Dessert

  - ''cake''     -- waist begins again
  - ''cream''    -- fluffy, white stuff
  - ''espresso'' -- life begins again

Current value: espresso
```

19.5 Dispatchers

The whole idea of a GlobalOptions class is that the options change the default behaviour of the associated classes. This can be done either by simply checking what the current value of the relevant option is. Another possibility is to use the options class as a dispatcher to associated methods. To use the dispatcher feature of a GlobalOptions class it is necessary to implement separate methods for each value of the option where the naming convention for these methods is that they start with a common prefix and finish with the value of the option.

If the value of a dispatchable option is set equal to a (user defined) function then this function is called instead of a class method.

For example, the options MyOptions can be used to dispatch the _repr_ method of the associated class MyClass as follows:

```
class MyClass(...):
    global_options=MyOptions
    def _repr_(self):
        return self.global_options.dispatch(self,'_repr_','first_option')
    def _repr_with_bells(self):
        print 'Bell!'
    def _repr_with_whistles(self):
        print 'Whistles!'
```

In this example, first_option is an option of MyOptions which takes values bells, whistles, and so on. Note that it is necessary to make self, which is an instance of MyClass, an argument of the dispatcher because dispatch() is a method of GlobalOptions and not a method of MyClass. Apart from MyOptions, as it is a method of this class, the arguments are the attached class (here MyClass), the prefix of the method of MyClass being dispatched, the option of MyOptions which controls the dispatching. All other arguments are passed through to the corresponding methods of MyClass. In general, a dispatcher is invoked as:

```
self.options.dispatch(self, dispatch_to, option, *args, **kargs)
```

Usually this will result in the method dispatch_to + '_' + MyOptions(options) of self being called with arguments *args and **kargs (if dispatch_to[-1] == '_' then the method dispatch_to + MyOptions(options) is called).

If MyOptions(options) is itself a function then the dispatcher will call this function instead. In this way, it is possible to allow the user to customise the default behaviour of this method. See dispatch() for an example of how this can be achieved.

The dispatching capabilities of GlobalOptions allows options to be applied automatically without needing to parse different values of the option (the cost is that there must be a method for each value). The dispatching capabilities can also be used to make one option control several methods:

```
def __le__(self, other):
    return self.options.dispatch(self, '_le_','cmp', other)
def __ge__(self, other):
    return self.options.dispatch(self, '_ge_','cmp', other)
def _le_option_a(self, other):
    return ...
def _ge_option_a(self, other):
    return ...
def _le_option_b(self, other):
    return ...
def _ge_option_b(self, other):
    return ...
```

See dispatch() for more details.

19.6 Doc testing

All of the options and their effects should be doc-tested. However, in order not to break other tests, all options should be returned to their default state at the end of each test. To make this easier, every `GlobalOptions` class has a `reset()` method for doing exactly this.

19.7 Tests

TESTS:

As options classes to not know how they are created they cannot be pickled:

```
sage: menu=GlobalOptions('menu', doc='Fancy documentation\n'+'-'*19, end_doc='The END!',
...       entree=dict(default='soup',
...                   description='The first course of a meal',
...                   values=dict(soup='soup of the day', bread='oven baked'),
...                   alias=dict(rye='bread')),
...       appetizer=dict(alt_name='entree'),
...       main=dict(default='pizza', description='Main meal',
...                 values=dict(pizza='thick crust', pasta='penne arrabiata'),
...                 case_sensitive=False),
...       dessert=dict(default='espresso', description='Dessert',
...                    values=dict(espresso='life begins again',
...                                cake='waist begins again',
...                                cream='fluffy, white stuff')),
...       tip=dict(default=10, description='Reward for good service',
...       checker=lambda tip: tip in range(0,20))
...   )
sage: TestSuite(menu).run(skip='_test_pickling')
```

Warning: Default values for `GlobalOptions` can be automatically overridden by calling the individual instances of the `GlobalOptions` class inside `$HOME/.sage/init.sage`. However, this needs to be disabled by developers when they are writing or checking doc-tests. Another possibly would be to `reset()` all options before and after all doct-tests which are dependent on particular values of options.

AUTHORS:

- Andrew Mathas (2013): initial version

class `sage.structure.global_options.`**`GlobalOptions`**(*name*, *doc*='', *end_doc*='', ***options*)
Bases: `sage.structure.sage_object.SageObject`

The `GlobalOptions` class is a generic class for setting and accessing global options for `sage` objects. It takes as inputs a `name` for the collection of options and a dictionary of dictionaries which specifies the individual options. The allowed/expected keys in the dictionary are the following:

INPUTS:

- `name` – Specifies a name for the options class (required)
- `doc` – Initial documentation string
- `end_doc` – Final documentation string
- `<options>=dict(...)` – Dictionary specifying an option

The options are specified by keyword arguments with their values being a dictionary which describes the option. The allowed/expected keys in the dictionary are:

- alias – Defines alias/synonym for option values
- alt_name – Alternative name for an option
- checker – A function for checking whether a particular value for the option is valid
- default – The default value of the option
- description – Documentation string
- link_to – Links to an option for this set of options to an option in another GlobalOptions
- setter – A function (class method) which is called whenever this option changes
- values – A dictionary of the legal values for this option (this automatically defines the corresponding checker). This dictionary gives the possible options, as keys, together with a brief description of them.
- case_sensitive – (Default: True) True or False depending on whether the values of the option are case sensitive.

Options and their values can be abbreviated provided that this abbreviation is a prefix of a unique option.

Calling the options with no arguments results in the list of current options being printed.

EXAMPLES:

```
sage: from sage.structure.global_options import GlobalOptions
sage: menu=GlobalOptions('menu', doc='Fancy documentation\n'+'-'*19, end_doc='End of Fancy docum
...     entree=dict(default='soup',
...                 description='The first course of a meal',
...                 values=dict(soup='soup of the day', bread='oven baked'),
...                 alias=dict(rye='bread')),
...     appetizer=dict(alt_name='entree'),
...     main=dict(default='pizza', description='Main meal',
...               values=dict(pizza='thick crust', pasta='penne arrabiata'),
...               case_sensitive=False),
...     dessert=dict(default='espresso', description='Dessert',
...                  values=dict(espresso='life begins again',
...                              cake='waist begins again',
...                              cream='fluffy white stuff')),
...     tip=dict(default=10, description='Reward for good service',
...              checker=lambda tip: tip in range(0,20))
... )
sage: menu
options for menu
sage: menu(entree='s')          # unambiguous abbreviations are allowed
sage: menu(t=15);
sage: (menu['tip'], menu('t'))
(15, 15)
sage: menu()
Current options for menu
  - dessert: espresso
  - entree:  soup
  - main:    pizza
  - tip:     15
sage: menu.reset(); menu()
Current options for menu
  - dessert: espresso
  - entree:  soup
  - main:    pizza
  - tip:     10
sage: menu['tip']=40
Traceback (most recent call last):
```

```
...
ValueError: 40 is not a valid value for tip in the options for menu
sage: menu(m='p')           # ambiguous abbreviations are not allowed
Traceback (most recent call last):
...
ValueError: p is not a valid value for main in the options for menu
```

The documentation for the options class is automatically generated from the information which specifies the options:

```
Fancy documentation
-------------------

OPTIONS:

- dessert:  (default: espresso)
  Dessert

  - ``cake``     -- waist begins again
  - ``cream``    -- fluffy white stuff
  - ``espresso`` -- life begins again

- entree:  (default: soup)
  The first course of a meal

  - ``bread`` -- oven baked
  - ``rye``   -- alias for bread
  - ``soup``  -- soup of the day

- main:  (default: pizza)
  Main meal

  - ``pasta`` -- penne arrabiata
  - ``pizza`` -- thick crust

- tip:  (default: 10)
  Reward for good service

End of Fancy documentation

See :class:`~sage.structure.global_options.GlobalOptions` for more features of these options.
```

The possible values for an individual option can be obtained by (trying to) set it equal to '?':

```
sage: menu(des='?')
- ``dessert`` -- (default: ``espresso``)
  Dessert

  - ``cake``     -- waist begins again
  - ``cream``    -- fluffy white stuff
  - ``espresso`` -- life begins again

Current value: espresso
```

default_value(*option*)

Return the default value of the option.

EXAMPLES:

```
sage: from sage.structure.global_options import GlobalOptions
sage: FoodOptions=GlobalOptions('daily meal',
...           food=dict(default='apple', values=dict(apple='a fruit',pair='of what?')),
...           drink=dict(default='water', values=dict(water='a drink',wine='a lifestyle')))
sage: FoodOptions.default_value('food')
'apple'
```

dispatch (*obj*, *dispatch_to*, *option*, **args*, ***kargs*)

Todo

title

The *dispatchable* options are options which dispatch related methods of the corresponding class - or user defined methods which are passed to `GlobalOptions`. The format for specifying a dispatchable option is to include `dispatch_to = <option name>` in the specifications for the options and then to add the options to the (element) class. Each option is then assumed to be a method of the element class with a name of the form `<option name> + '_' + <current vale for this option'`. These options are called by the element class via:

```
return self.options.dispatch(self, dispatch_to, option, *args, **kargs)
```

Note that the argument `self` is necessary here because the dispatcher is a method of the options class and not of `self`. The value of the option can also be set to a user-defined function, with arguments `self` and `option`; in this case the user's function is called instead.

EXAMPLES:

Here is a contrived example:

```
sage: from sage.structure.global_options import GlobalOptions
sage: DelimitedListOptions=GlobalOptions('list delimiters',
...             delim=dict(default='b', values={'b':'brackets', 'p':'parentheses'}))
sage: class DelimitedList(CombinatorialObject):
...      options=DelimitedListOptions
...      def _repr_b(self): return '[%s]' % ','.join('%s'%i for i in self._list)
...      def _repr_p(self): return '(%s)' % ','.join('%s'%i for i in self._list)
...      def _repr_(self): return self.options.dispatch(self, '_repr_','delim')
sage: dlist=DelimitedList([1,2,3]); dlist
[1,2,3]
sage: dlist.options(delim='p'); dlist
(1,2,3)
sage: dlist.options(delim=lambda self: '<%s>' % ','.join('%s'%i for i in self._list)); dlist
<1,2,3>
```

reset (*option=None*)

Reset options to their default value.

INPUT:

- `option` – (Default: `None`) The name of an option as a string or `None`. If `option` is specified only this option is reset to its default value; otherwise all options are reset.

EXAMPLES:

```
sage: from sage.structure.global_options import GlobalOptions
sage: opts=GlobalOptions('daily meal',
...      food=dict(default='bread', values=dict(bread='rye bread', salmon='a fish')),
...      drink=dict(default='water',values=dict(water='essential for life',wine='essential')
```

```
sage: opts(food='salmon'); opts()
Current options for daily meal
  - drink: water
  - food:  salmon
sage: opts.reset('drink'); opts()
Current options for daily meal
  - drink: water
  - food:  salmon
sage: opts.reset(); opts()
Current options for daily meal
  - drink: water
  - food:  bread
```

CHAPTER

TWENTY

CARTESIAN PRODUCTS

AUTHORS:

- Nicolas Thiery (2010-03): initial version

class `sage.sets.cartesian_product.CartesianProduct` (*sets*, *category*, *flatten=False*)

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.Parent`

A class implementing a raw data structure for cartesian products of sets (and elements thereof). See `cartesian_product` for how to construct full fledge cartesian products.

`_cartesian_product_of_elements` (*elements*)

Return the cartesian product of the given `elements`.

This implements `Sets.CartesianProducts.ParentMethods._cartesian_product_of_elements()`.

INPUT:

- `elements` – a tuple (or iterable) with one element of each cartesian factor of `self`

Warning: This is meant as a fast low-level method. In particular, no coercion is attempted. When coercion or sanity checks are desirable, please use instead `self(elements)` or `self._element_constructor(elements)`.

EXAMPLES:

```
sage: S1 = Sets().example()
sage: S2 = InfiniteEnumeratedSets().example()
sage: C = cartesian_product([S2, S1, S2])
sage: C._cartesian_product_of_elements([S2.an_element(), S1.an_element(), S2.an_element()])
(42, 47, 42)
```

class `Element`

Bases: `sage.structure.element_wrapper.ElementWrapper`

EXAMPLES:

```
sage: from sage.structure.element_wrapper import DummyParent
sage: a = ElementWrapper(DummyParent("A parent"), 1)
```

TESTS:

```
sage: TestSuite(a).run(skip = "_test_category")

sage: a = ElementWrapper(1, DummyParent("A parent"))
doctest:...: DeprecationWarning: the first argument must be a parent
See http://trac.sagemath.org/14519 for details.
```

Note: ElementWrapper is not intended to be used directly, hence the failing category test.

cartesian_projection(*i*)

Return the projection of self on the i-th factor of the cartesian product, as per Sets.CartesianProducts.ElementMethods.cartesian_projection().

INPUTS:

- i – the index of a factor of the cartesian product

EXAMPLES:

```
sage: C = Sets().CartesianProducts().example(); C
The cartesian product of (Set of prime numbers (basic implementation), An example of an i
sage: x = C.an_element(); x
(47, 42, 1)
sage: x.cartesian_projection(1)
42

sage: x.summand_projection(1)
doctest:...: DeprecationWarning: summand_projection is deprecated. Please use cartesian_p
See http://trac.sagemath.org/10963 for details.
42
```

CartesianProduct.**an_element**()

EXAMPLES:

```
sage: C = Sets().CartesianProducts().example(); C
The cartesian product of (Set of prime numbers (basic implementation),
 An example of an infinite enumerated set: the non negative integers,
 An example of a finite enumerated set: {1,2,3})
sage: C.an_element()
(47, 42, 1)
```

CartesianProduct.**cartesian_factors**()

Return the cartesian factors of self.

See also:

Sets.CartesianProducts.ParentMethods.cartesian_factors().

EXAMPLES:

```
sage: cartesian_product([QQ, ZZ, ZZ]).cartesian_factors()
(Rational Field, Integer Ring, Integer Ring)
```

CartesianProduct.**cartesian_projection**(*i*)

Return the natural projection onto the i-th cartesian factor of self as per Sets.CartesianProducts.ParentMethods.cartesian_projection().

INPUT:

- i – the index of a cartesian factor of self

EXAMPLES:

```
sage: C = Sets().CartesianProducts().example(); C
The cartesian product of (Set of prime numbers (basic implementation), An example of an infi
sage: x = C.an_element(); x
(47, 42, 1)
sage: pi = C.cartesian_projection(1)
sage: pi(x)
42
```

`CartesianProduct.`**`summand_projection`**(**args*, ***kwds*)

Deprecated: Use `cartesian_projection()` instead. See trac ticket #10963 for details.

CHAPTER

TWENTYONE

FAMILIES

A Family is an associative container which models a family $(f_i)_{i \in I}$. Then, f[i] returns the element of the family indexed by i. Whenever available, set and combinatorial class operations (counting, iteration, listing) on the family are induced from those of the index set. Families should be created through the Family() function.

AUTHORS:

- Nicolas Thiery (2008-02): initial release
- Florent Hivert (2008-04): various fixes, cleanups and improvements.

TESTS:

Check for workaround trac ticket #12482 (shall be run in a fresh session):

```
sage: P = Partitions(3)
sage: Family(P, lambda x: x).category() # used to return ''enumerated sets''
Category of finite enumerated sets
sage: Family(P, lambda x: x).category()
Category of finite enumerated sets
```

class sage.sets.family.**AbstractFamily**

Bases: sage.structure.parent.Parent

The abstract class for family

Any family belongs to a class which inherits from AbstractFamily.

hidden_keys()

Returns the hidden keys of the family, if any.

EXAMPLES:

```
sage: f = Family({3: 'a', 4: 'b', 7: 'd'})
sage: f.hidden_keys()
[]
```

inverse_family()

Returns the inverse family, with keys and values exchanged. This presumes that there are no duplicate values in self.

This default implementation is not lazy and therefore will only work with not too big finite families. It is also cached for the same reason:

```
sage: Family({3: 'a', 4: 'b', 7: 'd'}).inverse_family()
Finite family {'a': 3, 'b': 4, 'd': 7}

sage: Family((3,4,7)).inverse_family()
Finite family {3: 0, 4: 1, 7: 2}
```

map (*f*, *name=None*)

Returns the family $(f(\texttt{self}[i]))_{i\in I}$, where I is the index set of self.

Todo

good name?

EXAMPLES:

```
sage: f = Family({3: 'a', 4: 'b', 7: 'd'})
sage: g = f.map(lambda x: x+'1')
sage: list(g)
['a1', 'b1', 'd1']
```

zip (*f*, *other*, *name=None*)

Given two families with same index set I (and same hidden keys if relevant), returns the family $(f(self[i], other[i]))_{i\in I}$

Todo

generalize to any number of families and merge with map?

EXAMPLES:

```
sage: f = Family({3: 'a', 4: 'b', 7: 'd'})
sage: g = Family({3: '1', 4: '2', 7: '3'})
sage: h = f.zip(lambda x,y: x+y, g)
sage: list(h)
['a1', 'b2', 'd3']
```

class sage.sets.family.**EnumeratedFamily** (*enumset*)

Bases: sage.sets.family.LazyFamily

EnumeratedFamily turns an enumerated set c into a family indexed by the set $\{0,\dots,|c|-1\}$.

Instances should be created via the Family() factory. See its documentation for examples and tests.

cardinality ()

Return the number of elements in self.

EXAMPLES:

```
sage: from sage.sets.family import EnumeratedFamily
sage: f = EnumeratedFamily(Permutations(3))
sage: f.cardinality()
6

sage: from sage.categories.examples.infinite_enumerated_sets import NonNegativeIntegers
sage: f = Family(NonNegativeIntegers())
sage: f.cardinality()
+Infinity
```

keys ()

Returns self's keys.

EXAMPLES:

```
sage: from sage.sets.family import EnumeratedFamily
sage: f = EnumeratedFamily(Permutations(3))
sage: f.keys()
Standard permutations of 3
```

```
sage: from sage.categories.examples.infinite_enumerated_sets import NonNegativeIntegers
sage: f = Family(NonNegativeIntegers())
sage: f.keys()
An example of an infinite enumerated set: the non negative integers
```

`sage.sets.family.`**`Family`**(*indices*, *function=None*, *hidden_keys=*[ ], *hidden_function=None*, *lazy=False*, *name=None*)

A Family is an associative container which models a family $(f_i)_{i\in I}$. Then, `f[i]` returns the element of the family indexed by i. Whenever available, set and combinatorial class operations (counting, iteration, listing) on the family are induced from those of the index set.

There are several available implementations (classes) for different usages; Family serves as a factory, and will create instances of the appropriate classes depending on its arguments.

INPUT:

- `indices` – the indices for the family
- `function` – (optional) the function f applied to all visible indices; the default is the identity function
- `hidden_keys` – (optional) a list of hidden indices that can be accessed through `my_family[i]`
- `hidden_function` – (optional) a function for the hidden indices
- `lazy` – boolean (default: `False`); whether the family is lazily created or not; if the indices are infinite, then this is automatically made `True`
- `name` – (optional) the name of the function; only used when the family is lazily created via a function

EXAMPLES:

In its simplest form, a list $l = [l_0, l_1, \ldots, l_\ell]$ or a tuple by itself is considered as the family $(l_i)_{i\in I}$ where I is the set $\{0, \ldots, \ell\}$ where ℓ is `len(l) - 1`. So `Family(l)` returns the corresponding family:

```
sage: f = Family([1,2,3])
sage: f
Family (1, 2, 3)
sage: f = Family((1,2,3))
sage: f
Family (1, 2, 3)
```

Instead of a list you can as well pass any iterable object:

```
sage: f = Family(2*i+1 for i in [1,2,3]);
sage: f
Family (3, 5, 7)
```

A family can also be constructed from a dictionary `t`. The resulting family is very close to `t`, except that the elements of the family are the values of `t`. Here, we define the family $(f_i)_{i\in\{3,4,7\}}$ with $f_3 = a$, $f_4 = b$, and $f_7 = d$:

```
sage: f = Family({3: 'a', 4: 'b', 7: 'd'})
sage: f
Finite family {3: 'a', 4: 'b', 7: 'd'}
sage: f[7]
'd'
sage: len(f)
3
sage: list(f)
['a', 'b', 'd']
sage: [ x for x in f ]
['a', 'b', 'd']
```

```
sage: f.keys()
[3, 4, 7]
sage: 'b' in f
True
sage: 'e' in f
False
```

A family can also be constructed by its index set I and a function f, as in $(f(i))_{i \in I}$:

```
sage: f = Family([3,4,7], lambda i: 2*i)
sage: f
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
[3, 4, 7]
sage: f[7]
14
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
sage: len(f)
3
```

By default, all images are computed right away, and stored in an internal dictionary:

```
sage: f = Family((3,4,7), lambda i: 2*i)
sage: f
Finite family {3: 6, 4: 8, 7: 14}
```

Note that this requires all the elements of the list to be hashable. One can ask instead for the images $f(i)$ to be computed lazily, when needed:

```
sage: f = Family([3,4,7], lambda i: 2*i, lazy=True)
sage: f
Lazy family (<lambda>(i))_{i in [3, 4, 7]}
sage: f[7]
14
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
```

This allows in particular for modeling infinite families:

```
sage: f = Family(ZZ, lambda i: 2*i, lazy=True)
sage: f
Lazy family (<lambda>(i))_{i in Integer Ring}
sage: f.keys()
Integer Ring
sage: f[1]
2
sage: f[-5]
-10
sage: i = iter(f)
sage: next(i), next(i), next(i), next(i), next(i)
(0, 2, -2, 4, -4)
```

Note that the `lazy` keyword parameter is only needed to force laziness. Usually it is automatically set to a correct default value (ie: `False` for finite data structures and `True` for enumerated sets:

```
sage: f == Family(ZZ, lambda i: 2*i)
True
```

Beware that for those kind of families len(f) is not supposed to work. As a replacement, use the .cardinality() method:

```
sage: f = Family(Permutations(3), attrcall("to_lehmer_code"))
sage: list(f)
[[0, 0, 0], [0, 1, 0], [1, 0, 0], [1, 1, 0], [2, 0, 0], [2, 1, 0]]
sage: f.cardinality()
6
```

Caveat: Only certain families with lazy behavior can be pickled. In particular, only functions that work with Sage's pickle_function and unpickle_function (in sage.misc.fpickle) will correctly unpickle. The following two work:

```
sage: f = Family(Permutations(3), lambda p: p.to_lehmer_code()); f
Lazy family (<lambda>(i))_{i in Standard permutations of 3}
sage: f == loads(dumps(f))
True

sage: f = Family(Permutations(3), attrcall("to_lehmer_code")); f
Lazy family (i.to_lehmer_code())_{i in Standard permutations of 3}
sage: f == loads(dumps(f))
True
```

But this one does not:

```
sage: def plus_n(n): return lambda x: x+n
sage: f = Family([1,2,3], plus_n(3), lazy=True); f
Lazy family (<lambda>(i))_{i in [1, 2, 3]}
sage: f == loads(dumps(f))
Traceback (most recent call last):
...
ValueError: Cannot pickle code objects from closures
```

Finally, it can occasionally be useful to add some hidden elements in a family, which are accessible as `f[i]`, but do not appear in the keys or the container operations:

```
sage: f = Family([3,4,7], lambda i: 2*i, hidden_keys=[2])
sage: f
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
[3, 4, 7]
sage: f.hidden_keys()
[2]
sage: f[7]
14
sage: f[2]
4
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
sage: len(f)
3
```

The following example illustrates when the function is actually called:

```
sage: def compute_value(i):
...       print('computing 2*'+str(i))
...       return 2*i
sage: f = Family([3,4,7], compute_value, hidden_keys=[2])
computing 2*3
computing 2*4
computing 2*7
sage: f
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
[3, 4, 7]
sage: f.hidden_keys()
[2]
sage: f[7]
14
sage: f[2]
computing 2*2
4
sage: f[2]
4
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
sage: len(f)
3
```

Here is a close variant where the function for the hidden keys is different from that for the other keys:

```
sage: f = Family([3,4,7], lambda i: 2*i, hidden_keys=[2], hidden_function = lambda i: 3*i)
sage: f
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
[3, 4, 7]
sage: f.hidden_keys()
[2]
sage: f[7]
14
sage: f[2]
6
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
sage: len(f)
3
```

Family accept finite and infinite EnumeratedSets as input:

```
sage: f = Family(FiniteEnumeratedSet([1,2,3]))
sage: f
Family (1, 2, 3)
sage: from sage.categories.examples.infinite_enumerated_sets import NonNegativeIntegers
sage: f = Family(NonNegativeIntegers())
sage: f
Family (An example of an infinite enumerated set: the non negative integers)

sage: f = Family(FiniteEnumeratedSet([3,4,7]), lambda i: 2*i)
sage: f
```

```
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
{3, 4, 7}
sage: f[7]
14
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
sage: len(f)
3
```

TESTS:

```
sage: f = Family({1:'a', 2:'b', 3:'c'})
sage: f
Finite family {1: 'a', 2: 'b', 3: 'c'}
sage: f[2]
'b'
sage: loads(dumps(f)) == f
True

sage: f = Family({1:'a', 2:'b', 3:'c'}, lazy=True)
Traceback (most recent call last):
ValueError: lazy keyword only makes sense together with function keyword !

sage: f = Family(range(1,27), lambda i: chr(i+96))
sage: f
    Finite family {1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e', 6: 'f', 7: 'g', 8: 'h', 9: 'i', 10: '
sage: f[2]
'b'
```

The factory `Family` is supposed to be idempotent. We test this feature here:

```
sage: from sage.sets.family import FiniteFamily, LazyFamily, TrivialFamily
sage: f = FiniteFamily({3: 'a', 4: 'b', 7: 'd'})
sage: g = Family(f)
sage: f == g
True

sage: f = Family([3,4,7], lambda i: 2*i, hidden_keys=[2])
sage: g = Family(f)
sage: f == g
True

sage: f = LazyFamily([3,4,7], lambda i: 2*i)
sage: g = Family(f)
sage: f == g
True

sage: f = TrivialFamily([3,4,7])
sage: g = Family(f)
sage: f == g
True
```

The family should keep the order of the keys:

```
sage: f = Family(["c", "a", "b"], lambda x: x+x)
sage: list(f)
['cc', 'aa', 'bb']
```

TESTS:

Only the hidden function is applied to the hidden keys:

```
sage: f = lambda x : 2*x
sage: h_f = lambda x:x%2
sage: F = Family([1,2,3,4],function = f, hidden_keys=[5],hidden_function=h_f)
sage: F[5]
1
```

class sage.sets.family.**FiniteFamily**(*dictionary*, *keys=None*)

Bases: sage.sets.family.AbstractFamily

A FiniteFamily is an associative container which models a finite family $(f_i)_{i\in I}$. Its elements f_i are therefore its values. Instances should be created via the Family() factory. See its documentation for examples and tests.

EXAMPLES:

We define the family $(f_i)_{i\in\{3,4,7\}}$ with $f_3 = a$, $f_4 = b$, and $f_7 = d$:

```
sage: from sage.sets.family import FiniteFamily
sage: f = FiniteFamily({3: 'a', 4: 'b', 7: 'd'})
```

Individual elements are accessible as in a usual dictionary:

```
sage: f[7]
'd'
```

And the other usual dictionary operations are also available:

```
sage: len(f)
3
sage: f.keys()
[3, 4, 7]
```

However f behaves as a container for the f_i's:

```
sage: list(f)
['a', 'b', 'd']
sage: [ x for x in f ]
['a', 'b', 'd']
```

The order of the elements can be specified using the `keys` optional argument:

```
sage: f = FiniteFamily({"a": "aa", "b": "bb", "c" : "cc" }, keys = ["c", "a", "b"])
sage: list(f)
['cc', 'aa', 'bb']
```

cardinality()

Returns the number of elements in self.

EXAMPLES:

```
sage: from sage.sets.family import FiniteFamily
sage: f = FiniteFamily({3: 'a', 4: 'b', 7: 'd'})
sage: f.cardinality()
3
```

has_key(*k*)

Returns whether `k` is a key of `self`

EXAMPLES:

```
sage: Family({"a":1, "b":2, "c":3}).has_key("a")
True
sage: Family({"a":1, "b":2, "c":3}).has_key("d")
False
```

keys()

Returns the index set of this family

EXAMPLES:

```
sage: f = Family(["c", "a", "b"], lambda x: x+x)
sage: f.keys()
['c', 'a', 'b']
```

values()

Returns the elements of this family

EXAMPLES:

```
sage: f = Family(["c", "a", "b"], lambda x: x+x)
sage: f.values()
['cc', 'aa', 'bb']
```

class sage.sets.family.**FiniteFamilyWithHiddenKeys**(*dictionary*, *hidden_keys*, *hidden_function*)

Bases: sage.sets.family.FiniteFamily

A close variant of FiniteFamily where the family contains some hidden keys whose corresponding values are computed lazily (and remembered). Instances should be created via the Family() factory. See its documentation for examples and tests.

Caveat: Only instances of this class whose functions are compatible with sage.misc.fpickle can be pickled.

hidden_keys()

Returns self's hidden keys.

EXAMPLES:

```
sage: f = Family([3,4,7], lambda i: 2*i, hidden_keys=[2])
sage: f.hidden_keys()
[2]
```

class sage.sets.family.**LazyFamily**(*set*, *function*, *name=None*)

Bases: sage.sets.family.AbstractFamily

A LazyFamily(I, f) is an associative container which models the (possibly infinite) family $(f(i))_{i \in I}$.

Instances should be created via the Family() factory. See its documentation for examples and tests.

cardinality()

Return the number of elements in self.

EXAMPLES:

```
sage: from sage.sets.family import LazyFamily
sage: f = LazyFamily([3,4,7], lambda i: 2*i)
sage: f.cardinality()
3
sage: from sage.categories.examples.infinite_enumerated_sets import NonNegativeIntegers
sage: l = LazyFamily(NonNegativeIntegers(), lambda i: 2*i)
```

```
sage: l.cardinality()
+Infinity
```

TESTS:

Check that trac ticket #15195 is fixed:

```
sage: C = CartesianProduct(PositiveIntegers(), [1,2,3])
sage: C.cardinality()
+Infinity
sage: F = Family(C, lambda x: x)
sage: F.cardinality()
+Infinity
```

keys()
Returns self's keys.

EXAMPLES:

```
sage: from sage.sets.family import LazyFamily
sage: f = LazyFamily([3,4,7], lambda i: 2*i)
sage: f.keys()
[3, 4, 7]
```

class sage.sets.family.**TrivialFamily**(*enumeration*)
Bases: sage.sets.family.AbstractFamily

TrivialFamily turns a list/tuple c into a family indexed by the set $\{0, \dots, |c| - 1\}$.

Instances should be created via the Family() factory. See its documentation for examples and tests.

cardinality()
Return the number of elements in self.

EXAMPLES:

```
sage: from sage.sets.family import TrivialFamily
sage: f = TrivialFamily([3,4,7])
sage: f.cardinality()
3
```

keys()
Returns self's keys.

EXAMPLES:

```
sage: from sage.sets.family import TrivialFamily
sage: f = TrivialFamily([3,4,7])
sage: f.keys()
[0, 1, 2]
```

CHAPTER

TWENTYTWO

SETS

AUTHORS:

- William Stein (2005) - first version
- William Stein (2006-02-16) - large number of documentation and examples; improved code
- Mike Hansen (2007-3-25) - added differences and symmetric differences; fixed operators
- Florent Hivert (2010-06-17) - Adapted to categories
- Nicolas M. Thiery (2011-03-15) - Added subset and superset methods
- Julian Rueth (2013-04-09) - Collected common code in `Set_object_binary`, fixed `__hash__`.

`sage.sets.set.Set` (*X=frozenset([])*)

Create the underlying set of `X`.

If `X` is a list, tuple, Python set, or `X.is_finite()` is `True`, this returns a wrapper around Python's enumerated immutable `frozenset` type with extra functionality. Otherwise it returns a more formal wrapper.

If you need the functionality of mutable sets, use Python's builtin set type.

EXAMPLES:

```
sage: X = Set(GF(9,'a'))
sage: X
{0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2}
sage: type(X)
<class 'sage.sets.set.Set_object_enumerated_with_category'>
sage: Y = X.union(Set(QQ))
sage: Y
Set-theoretic union of {0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2} and Set of elements of
sage: type(Y)
<class 'sage.sets.set.Set_object_union_with_category'>
```

Usually sets can be used as dictionary keys.

```
sage: d={Set([2*I,1+I]):10}
sage: d                  # key is randomly ordered
{{I + 1, 2*I}: 10}
sage: d[Set([1+I,2*I])]
10
sage: d[Set((1+I,2*I))]
10
```

The original object is often forgotten.

```
sage: v = [1,2,3]
sage: X = Set(v)
```

```
sage: X
{1, 2, 3}
sage: v.append(5)
sage: X
{1, 2, 3}
sage: 5 in X
False
```

Set also accepts iterators, but be careful to only give *finite* sets.

```
sage: list(Set(iter([1, 2, 3, 4, 5])))
[1, 2, 3, 4, 5]
```

We can also create sets from different types:

```
sage: sorted(Set([Sequence([3,1], immutable=True), 5, QQ, Partition([3,1,1])]), key=str)
[5, Rational Field, [3, 1, 1], [3, 1]]
```

However each of the objects must be hashable:

```
sage: Set([QQ, [3, 1], 5])
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

TESTS:

```
sage: Set(Primes())
Set of all prime numbers: 2, 3, 5, 7, ...
sage: Set(Subsets([1,2,3])).cardinality()
8
sage: S = Set(iter([1,2,3])); S
{1, 2, 3}
sage: type(S)
<class 'sage.sets.set.Set_object_enumerated_with_category'>
sage: S = Set([])
sage: TestSuite(S).run()
```

Check that trac ticket #16090 is fixed:

```
sage: Set()
{}
```

class sage.sets.set.**Set_object**(*X*)

Bases: sage.structure.parent.Set_generic

A set attached to an almost arbitrary object.

EXAMPLES:

```
sage: K = GF(19)
sage: Set(K)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}
sage: S = Set(K)

sage: latex(S)
\left\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18\right\}
sage: TestSuite(S).run()

sage: latex(Set(ZZ))
\Bold{Z}
```

TESTS:

See trac ticket trac ticket #14486:

```
sage: 0 == Set([1]), Set([1]) == 0
(False, False)
sage: 1 == Set([0]), Set([0]) == 1
(False, False)
```

an_element()

Returns the first element of `self` returned by `__iter__()`

If `self` is empty, the exception `EmptySetError` is raised instead.

This provides a generic implementation of the method `_an_element_()` for all parents in `EnumeratedSets`.

EXAMPLES:

```
sage: C = FiniteEnumeratedSets().example(); C
An example of a finite enumerated set: {1,2,3}
sage: C.an_element() # indirect doctest
1
sage: S = Set([])
sage: S.an_element()
Traceback (most recent call last):
...
EmptySetError
```

TESTS:

```
sage: super(Parent, C)._an_element_
Cached version of <function _an_element_from_iterator at ...>
```

cardinality()

Return the cardinality of this set, which is either an integer or `Infinity`.

EXAMPLES:

```
sage: Set(ZZ).cardinality()
+Infinity
sage: Primes().cardinality()
+Infinity
sage: Set(GF(5)).cardinality()
5
sage: Set(GF(5^2,'a')).cardinality()
25
```

difference(*X*)

Return the set difference `self - X`.

EXAMPLES:

```
sage: X = Set(ZZ).difference(Primes())
sage: 4 in X
True
sage: 3 in X
False

sage: 4/1 in X
True

sage: X = Set(GF(9,'b')).difference(Set(GF(27,'c')))
```

```
sage: X
{0, 1, 2, b, b + 1, b + 2, 2*b, 2*b + 1, 2*b + 2}

sage: X = Set(GF(9,'b')).difference(Set(GF(27,'b')))
sage: X
{0, 1, 2, b, b + 1, b + 2, 2*b, 2*b + 1, 2*b + 2}
```

intersection(*X*)

Return the intersection of `self` and `X`.

EXAMPLES:

```
sage: X = Set(ZZ).intersection(Primes())
sage: 4 in X
False
sage: 3 in X
True

sage: 2/1 in X
True

sage: X = Set(GF(9,'b')).intersection(Set(GF(27,'c')))
sage: X
{}

sage: X = Set(GF(9,'b')).intersection(Set(GF(27,'b')))
sage: X
{}
```

is_empty()

Return boolean representing emptiness of the set.

OUTPUT:

True if the set is empty, false if otherwise.

EXAMPLES:

```
sage: Set([]).is_empty()
True
sage: Set([0]).is_empty()
False
sage: Set([1..100]).is_empty()
False
sage: Set(SymmetricGroup(2).list()).is_empty()
False
sage: Set(ZZ).is_empty()
False
```

TESTS:

```
sage: Set([]).is_empty()
True
sage: Set([1,2,3]).is_empty()
False
sage: Set([1..100]).is_empty()
False
sage: Set(DihedralGroup(4).list()).is_empty()
False
sage: Set(QQ).is_empty()
False
```

is_finite()

Return True if self is finite.

EXAMPLES:

```
sage: Set(QQ).is_finite()
False
sage: Set(GF(250037)).is_finite()
True
sage: Set(Integers(2^1000000)).is_finite()
True
sage: Set([1,'a',ZZ]).is_finite()
True
```

object()

Return underlying object.

EXAMPLES:

```
sage: X = Set(QQ)
sage: X.object()
Rational Field
sage: X = Primes()
sage: X.object()
Set of all prime numbers: 2, 3, 5, 7, ...
```

subsets(*size=None*)

Return the Subsets object representing the subsets of a set. If size is specified, return the subsets of that size.

EXAMPLES:

```
sage: X = Set([1,2,3])
sage: list(X.subsets())
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
sage: list(X.subsets(2))
[{1, 2}, {1, 3}, {2, 3}]
```

symmetric_difference(*X*)

Returns the symmetric difference of self and X.

EXAMPLES:

```
sage: X = Set([1,2,3]).symmetric_difference(Set([3,4]))
sage: X
{1, 2, 4}
```

union(*X*)

Return the union of self and X.

EXAMPLES:

```
sage: Set(QQ).union(Set(ZZ))
Set-theoretic union of Set of elements of Rational Field and Set of elements of Integer Ring
sage: Set(QQ) + Set(ZZ)
Set-theoretic union of Set of elements of Rational Field and Set of elements of Integer Ring
sage: X = Set(QQ).union(Set(GF(3))); X
Set-theoretic union of Set of elements of Rational Field and {0, 1, 2}
sage: 2/3 in X
True
```

```
sage: GF(3)(2) in X
True
sage: GF(5)(2) in X
False
sage: Set(GF(7)) + Set(GF(3))
{0, 1, 2, 3, 4, 5, 6, 1, 2, 0}
```

class sage.sets.set.**Set_object_binary**(*X*, *Y*, *op*, *latex_op*)

Bases: sage.sets.set.Set_object

An abstract common base class for sets defined by a binary operation (ex. Set_object_union, Set_object_intersection, Set_object_difference, and Set_object_symmetric_difference).

INPUT:

- X, Y – sets, the operands to op
- op – a string describing the binary operation
- latex_op – a string used for rendering this object in LaTeX

EXAMPLES:

```
sage: X = Set(QQ^2)
sage: Y = Set(ZZ)
sage: from sage.sets.set import Set_object_binary
sage: S = Set_object_binary(X, Y, "union", "\\cup"); S
Set-theoretic union of Set of elements of Vector space of dimension 2
 over Rational Field and Set of elements of Integer Ring
```

cardinality()

This tries to return the cardinality of this set.

Note that this is not likely to work in very much generality, and may just hang if either set involved is infinite.

EXAMPLES:

```
sage: X = Set(GF(13)).intersection(Set(ZZ))
sage: X.cardinality()
13
```

class sage.sets.set.**Set_object_difference**(*X*, *Y*)

Bases: sage.sets.set.Set_object_binary

Formal difference of two sets.

class sage.sets.set.**Set_object_enumerated**(*X*)

Bases: sage.sets.set.Set_object

A finite enumerated set.

cardinality()

Return the cardinality of self.

EXAMPLES:

```
sage: Set([1,1]).cardinality()
1
```

difference(*other*)

Return the set difference self - other.

EXAMPLES:

```
sage: X = Set([1,2,3,4])
sage: Y = Set([1,2])
sage: X.difference(Y)
{3, 4}
sage: Z = Set(ZZ)
sage: W = Set([2.5, 4, 5, 6])
sage: W.difference(Z)
{2.50000000000000}
```

frozenset()

Return the Python frozenset object associated to this set, which is an immutable set (hence hashable).

EXAMPLES:

```
sage: X = Set(GF(8,'c'))
sage: X
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: s = X.set(); s
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: hash(s)
Traceback (most recent call last):
...
TypeError: unhashable type: 'set'
sage: s = X.frozenset(); s
frozenset({0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1})
sage: hash(s)
-1390224788            # 32-bit
 561411537695332972    # 64-bit
sage: type(s)
<type 'frozenset'>
```

intersection(*other*)

Return the intersection of `self` and `other`.

EXAMPLES:

```
sage: X = Set(GF(8,'c'))
sage: Y = Set([GF(8,'c').0, 1, 2, 3])
sage: X.intersection(Y)
{1, c}
```

issubset(*other*)

Return whether `self` is a subset of `other`.

INPUT:

- `other` – a finite Set

EXAMPLES:

```
sage: X = Set([1,3,5])
sage: Y = Set([0,1,2,3,5,7])
sage: X.issubset(Y)
True
sage: Y.issubset(X)
False
sage: X.issubset(X)
True
```

TESTS:

```
sage: len([Z for Z in Y.subsets() if Z.issubset(X)])
8
```

issuperset(*other*)

Return whether `self` is a superset of `other`.

INPUT:

- `other` – a finite Set

EXAMPLES:

```
sage: X = Set([1,3,5])
sage: Y = Set([0,1,2,3,5])
sage: X.issuperset(Y)
False
sage: Y.issuperset(X)
True
sage: X.issuperset(X)
True
```

TESTS:

```
sage: len([Z for Z in Y.subsets() if Z.issuperset(X)])
4
```

list()

Return the elements of `self`, as a list.

EXAMPLES:

```
sage: X = Set(GF(8,'c'))
sage: X
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: X.list()
[0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1]
sage: type(X.list())
<type 'list'>
```

Todo

FIXME: What should be the order of the result? That of `self.object()`? Or the order given by `set(self.object())`? Note that `__getitem__()` is currently implemented in term of this list method, which is really inefficient ...

set()

Return the Python set object associated to this set.

Python has a notion of finite set, and often Sage sets have an associated Python set. This function returns that set.

EXAMPLES:

```
sage: X = Set(GF(8,'c'))
sage: X
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: X.set()
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: type(X.set())
<type 'set'>
```

```
sage: type(X)
<class 'sage.sets.set.Set_object_enumerated_with_category'>
```

symmetric_difference(*other*)

Return the symmetric difference of `self` and `other`.

EXAMPLES:

```
sage: X = Set([1,2,3,4])
sage: Y = Set([1,2])
sage: X.symmetric_difference(Y)
{3, 4}
sage: Z = Set(ZZ)
sage: W = Set([2.5, 4, 5, 6])
sage: U = W.symmetric_difference(Z)
sage: 2.5 in U
True
sage: 4 in U
False
sage: V = Z.symmetric_difference(W)
sage: V == U
True
sage: 2.5 in V
True
sage: 6 in V
False
```

union(*other*)

Return the union of `self` and `other`.

EXAMPLES:

```
sage: X = Set(GF(8,'c'))
sage: Y = Set([GF(8,'c').0, 1, 2, 3])
sage: X
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: Y
{1, c, 3, 2}
sage: X.union(Y)
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1, 2, 3}
```

class sage.sets.set.**Set_object_intersection**(*X*, *Y*)

Bases: sage.sets.set.Set_object_binary

Formal intersection of two sets.

class sage.sets.set.**Set_object_symmetric_difference**(*X*, *Y*)

Bases: sage.sets.set.Set_object_binary

Formal symmetric difference of two sets.

class sage.sets.set.**Set_object_union**(*X*, *Y*)

Bases: sage.sets.set.Set_object_binary

A formal union of two sets.

cardinality()

Return the cardinality of this set.

EXAMPLES:

```
sage: X = Set(GF(3)).union(Set(GF(2)))
sage: X
{0, 1, 2, 0, 1}
sage: X.cardinality()
5

sage: X = Set(GF(3)).union(Set(ZZ))
sage: X.cardinality()
+Infinity
```

sage.sets.set.**is_Set**(*x*)

Returns True if x is a Sage Set_object (not to be confused with a Python set).

EXAMPLES:

```
sage: from sage.sets.set import is_Set
sage: is_Set([1,2,3])
False
sage: is_Set(set([1,2,3]))
False
sage: is_Set(Set([1,2,3]))
True
sage: is_Set(Set(QQ))
True
sage: is_Set(Primes())
True
```

CHAPTER

TWENTYTHREE

DISJOINT-SET DATA STRUCTURE

The main entry point is `DisjointSet()` which chooses the appropriate type to return. For more on the data structure, see `DisjointSet()`.

AUTHORS:

- Sebastien Labbe (2008) - Initial version.
- Sebastien Labbe (2009-11-24) - Pickling support
- Sebastien Labbe (2010-01) - Inclusion into sage (trac ticket #6775).

EXAMPLES:

Disjoint set of integers from `0` to `n - 1`:

```
sage: s = DisjointSet(6)
sage: s
{{0}, {1}, {2}, {3}, {4}, {5}}
sage: s.union(2, 4)
sage: s.union(1, 3)
sage: s.union(5, 1)
sage: s
{{0}, {1, 3, 5}, {2, 4}}
sage: s.find(3)
1
sage: s.find(5)
1
sage: map(s.find, range(6))
[0, 1, 2, 1, 2, 1]
```

Disjoint set of hashables objects:

```
sage: d = DisjointSet('abcde')
sage: d
{{'a'}, {'b'}, {'c'}, {'d'}, {'e'}}
sage: d.union('a','b')
sage: d.union('b','c')
sage: d.union('c','d')
sage: d
{{'a', 'b', 'c', 'd'}, {'e'}}
sage: d.find('c')
'a'
```

`sage.sets.disjoint_set.`**`DisjointSet`**`(`*arg*`)`

Constructs a disjoint set where each element of `arg` is in its own set. If `arg` is an integer, then the disjoint set returned is made of the integers from `0` to `arg - 1`.

A disjoint-set data structure (sometimes called union-find data structure) is a data structure that keeps track of a partitioning of a set into a number of separate, nonoverlapping sets. It performs two operations:

- `find()` – Determine which set a particular element is in.
- `union()` – Combine or merge two sets into a single set.

REFERENCES:

- Wikipedia article Disjoint-set_data_structure

INPUT:

- `arg` – non negative integer or an iterable of hashable objects.

EXAMPLES:

From a non-negative integer:

```
sage: DisjointSet(5)
{{0}, {1}, {2}, {3}, {4}}
```

From an iterable:

```
sage: DisjointSet('abcde')
{{'a'}, {'b'}, {'c'}, {'d'}, {'e'}}
sage: DisjointSet(range(6))
{{0}, {1}, {2}, {3}, {4}, {5}}
sage: DisjointSet(['yi',45,'cheval'])
{{'cheval'}, {'yi'}, {45}}
```

TESTS:

```
sage: DisjointSet(0)
{}
sage: DisjointSet('')
{}
sage: DisjointSet([])
{}
```

The argument must be a non negative integer:

```
sage: DisjointSet(-1)
Traceback (most recent call last):
...
ValueError: arg (=-1) must be a non negative integer
```

or an iterable:

```
sage: DisjointSet(4.3)
Traceback (most recent call last):
...
TypeError: 'sage.rings.real_mpfr.RealLiteral' object is not iterable
```

Moreover, the iterable must consist of hashable:

```
sage: DisjointSet([{}, {}])
Traceback (most recent call last):
...
TypeError: unhashable type: 'dict'
```

class `sage.sets.disjoint_set.`**`DisjointSet_class`**

Bases: `sage.structure.sage_object.SageObject`

Common class and methods for `DisjointSet_of_integers` and `DisjointSet_of_hashables`.

cardinality()

Return the number of elements in `self`, *not* the number of subsets.

EXAMPLES:

```
sage: d = DisjointSet(5)
sage: d.cardinality()
5
sage: d.union(2, 4)
sage: d.cardinality()
5
sage: d = DisjointSet(range(5))
sage: d.cardinality()
5
sage: d.union(2, 4)
sage: d.cardinality()
5
```

number_of_subsets()

Return the number of subsets in `self`.

EXAMPLES:

```
sage: d = DisjointSet(5)
sage: d.number_of_subsets()
5
sage: d.union(2, 4)
sage: d.number_of_subsets()
4
sage: d = DisjointSet(range(5))
sage: d.number_of_subsets()
5
sage: d.union(2, 4)
sage: d.number_of_subsets()
4
```

class `sage.sets.disjoint_set.`**DisjointSet_of_hashables**

Bases: `sage.sets.disjoint_set.DisjointSet_class`

Disjoint set of hashables.

EXAMPLES:

```
sage: d = DisjointSet('abcde')
sage: d
{{'a'}, {'b'}, {'c'}, {'d'}, {'e'}}
sage: d.union('a', 'c')
sage: d
{{'a', 'c'}, {'b'}, {'d'}, {'e'}}
sage: d.find('a')
'a'
```

TESTS:

```
sage: a = DisjointSet('abcdef')
sage: a == loads(dumps(a))
True

sage: a.union('a','c')
sage: a == loads(dumps(a))
```

```
True
```

element_to_root_dict()

Return the dictionary where the keys are the elements of `self` and the values are their representative inside a list.

EXAMPLES:

```
sage: d = DisjointSet(range(5))
sage: d.union(2,3)
sage: d.union(4,1)
sage: e = d.element_to_root_dict(); e
{0: 0, 1: 4, 2: 2, 3: 2, 4: 4}
sage: WordMorphism(e)
WordMorphism: 0->0, 1->4, 2->2, 3->2, 4->4
```

find(*e*)

Return the representative of the set that `e` currently belongs to.

INPUT:

- `e` – element in `self`

EXAMPLES:

```
sage: e = DisjointSet(range(5))
sage: e.union(4,2)
sage: e
{{0}, {1}, {2, 4}, {3}}
sage: e.find(2)
4
sage: e.find(4)
4
sage: e.union(1,3)
sage: e
{{0}, {1, 3}, {2, 4}}
sage: e.find(1)
1
sage: e.find(3)
1
sage: e.union(3,2)
sage: e
{{0}, {1, 2, 3, 4}}
sage: [e.find(i) for i in range(5)]
[0, 1, 1, 1, 1]
sage: e.find(5)
Traceback (most recent call last):
...
KeyError: 5
```

root_to_elements_dict()

Return the dictionary where the keys are the roots of `self` and the values are the elements in the same set.

EXAMPLES:

```
sage: d = DisjointSet(range(5))
sage: d.union(2,3)
sage: d.union(4,1)
sage: e = d.root_to_elements_dict(); e
{0: [0], 2: [2, 3], 4: [1, 4]}
```

to_digraph()

Return the current digraph of `self` where (a, b) is an oriented edge if b is the parent of a.

EXAMPLES:

```
sage: d = DisjointSet(range(5))
sage: d.union(2,3)
sage: d.union(4,1)
sage: d.union(3,4)
sage: d
{{0}, {1, 2, 3, 4}}
sage: g = d.to_digraph(); g
Looped digraph on 5 vertices
sage: g.edges()
[(0, 0, None), (1, 2, None), (2, 2, None), (3, 2, None), (4, 2, None)]
```

The result depends on the ordering of the union:

```
sage: d = DisjointSet(range(5))
sage: d.union(1,2)
sage: d.union(1,3)
sage: d.union(1,4)
sage: d
{{0}, {1, 2, 3, 4}}
sage: d.to_digraph().edges()
[(0, 0, None), (1, 1, None), (2, 1, None), (3, 1, None), (4, 1, None)]
```

union(e, f)

Combine the set of `e` and the set of `f` into one.

All elements in those two sets will share the same representative that can be gotten using find.

INPUT:

- `e` - element in `self`
- `f` - element in `self`

EXAMPLES:

```
sage: e = DisjointSet('abcde')
sage: e
{{'a'}, {'b'}, {'c'}, {'d'}, {'e'}}
sage: e.union('a','b')
sage: e
{{'a', 'b'}, {'c'}, {'d'}, {'e'}}
sage: e.union('c','e')
sage: e
{{'a', 'b'}, {'c', 'e'}, {'d'}}
sage: e.union('b','e')
sage: e
{{'a', 'b', 'c', 'e'}, {'d'}}
```

class `sage.sets.disjoint_set.`**DisjointSet_of_integers**

Bases: `sage.sets.disjoint_set.DisjointSet_class`

Disjoint set of integers from `0` to `n-1`.

EXAMPLES:

```
sage: d = DisjointSet(5)
sage: d
{{0}, {1}, {2}, {3}, {4}}
```

```
sage: d.union(2,4)
sage: d.union(0,2)
sage: d
{{0, 2, 4}, {1}, {3}}
sage: d.find(2)
2
```

TESTS:

```
sage: a = DisjointSet(5)
sage: a == loads(dumps(a))
True

sage: a.union(3,4)
sage: a == loads(dumps(a))
True
```

element_to_root_dict()

Return the dictionary where the keys are the elements of `self` and the values are their representative inside a list.

EXAMPLES:

```
sage: d = DisjointSet(5)
sage: d.union(2,3)
sage: d.union(4,1)
sage: e = d.element_to_root_dict(); e
{0: 0, 1: 4, 2: 2, 3: 2, 4: 4}
sage: WordMorphism(e)
WordMorphism: 0->0, 1->4, 2->2, 3->2, 4->4
```

find(*i*)

Return the representative of the set that `i` currently belongs to.

INPUT:

- `i` – element in `self`

EXAMPLES:

```
sage: e = DisjointSet(5)
sage: e.union(4,2)
sage: e
{{0}, {1}, {2, 4}, {3}}
sage: e.find(2)
4
sage: e.find(4)
4
sage: e.union(1,3)
sage: e
{{0}, {1, 3}, {2, 4}}
sage: e.find(1)
1
sage: e.find(3)
1
sage: e.union(3,2)
sage: e
{{0}, {1, 2, 3, 4}}
sage: [e.find(i) for i in range(5)]
[0, 1, 1, 1, 1]
sage: e.find(5)
```

```
Traceback (most recent call last):
...
ValueError: i(=5) must be between 0 and 4
```

root_to_elements_dict()

Return the dictionary where the keys are the roots of `self` and the values are the elements in the same set as the root.

EXAMPLES:

```
sage: d = DisjointSet(5)
sage: d.root_to_elements_dict()
{0: [0], 1: [1], 2: [2], 3: [3], 4: [4]}
sage: d.union(2,3)
sage: d.root_to_elements_dict()
{0: [0], 1: [1], 2: [2, 3], 4: [4]}
sage: d.union(3,0)
sage: d.root_to_elements_dict()
{1: [1], 2: [0, 2, 3], 4: [4]}
sage: d
{{0, 2, 3}, {1}, {4}}
```

to_digraph()

Return the current digraph of `self` where (a, b) is an oriented edge if b is the parent of a.

EXAMPLES:

```
sage: d = DisjointSet(5)
sage: d.union(2,3)
sage: d.union(4,1)
sage: d.union(3,4)
sage: d
{{0}, {1, 2, 3, 4}}
sage: g = d.to_digraph(); g
Looped digraph on 5 vertices
sage: g.edges()
[(0, 0, None), (1, 2, None), (2, 2, None), (3, 2, None), (4, 2, None)]
```

The result depends on the ordering of the union:

```
sage: d = DisjointSet(5)
sage: d.union(1,2)
sage: d.union(1,3)
sage: d.union(1,4)
sage: d
{{0}, {1, 2, 3, 4}}
sage: d.to_digraph().edges()
[(0, 0, None), (1, 1, None), (2, 1, None), (3, 1, None), (4, 1, None)]
```

union(i, j)

Combine the set of `i` and the set of `j` into one.

All elements in those two sets will share the same representative that can be gotten using find.

INPUT:

- `i` - element in `self`
- `j` - element in `self`

EXAMPLES:

```
sage: d = DisjointSet(5)
sage: d
{{0}, {1}, {2}, {3}, {4}}
sage: d.union(0,1)
sage: d
{{0, 1}, {2}, {3}, {4}}
sage: d.union(2,4)
sage: d
{{0, 1}, {2, 4}, {3}}
sage: d.union(1,4)
sage: d
{{0, 1, 2, 4}, {3}}
sage: d.union(1,5)
Traceback (most recent call last):
...
ValueError: j(=5) must be between 0 and 4
```

sage.sets.disjoint_set.**OP_represent**(*n*, *merges*, *perm*)

Demonstration and testing.

TESTS:

```
sage: from sage.groups.perm_gps.partn_ref.automorphism_group_canonical_label import OP_represent
sage: OP_represent(9, [(0,1),(2,3),(3,4)], [1,2,0,4,3,6,7,5,8])
Allocating OrbitPartition...
Allocation passed.
Checking that each element reports itself as its root.
Each element reports itself as its root.
Merging:
Merged 0 and 1.
Merged 2 and 3.
Merged 3 and 4.
Done merging.
Finding:
0 -> 0, root: size=2, mcr=0, rank=1
1 -> 0
2 -> 2, root: size=3, mcr=2, rank=1
3 -> 2
4 -> 2
5 -> 5, root: size=1, mcr=5, rank=0
6 -> 6, root: size=1, mcr=6, rank=0
7 -> 7, root: size=1, mcr=7, rank=0
8 -> 8, root: size=1, mcr=8, rank=0
Allocating array to test merge_perm.
Allocation passed.
Merging permutation: [1, 2, 0, 4, 3, 6, 7, 5, 8]
Done merging.
Finding:
0 -> 0, root: size=5, mcr=0, rank=2
1 -> 0
2 -> 0
3 -> 0
4 -> 0
5 -> 5, root: size=3, mcr=5, rank=1
6 -> 5
7 -> 5
8 -> 8, root: size=1, mcr=8, rank=0
Deallocating OrbitPartition.
Done.
```

sage.sets.disjoint_set.**PS_represent**(*partition*, *splits*)

Demonstration and testing.

TESTS:

```
sage: from sage.groups.perm_gps.partn_ref.automorphism_group_canonical_label import PS_represent
sage: PS_represent([[6],[3,4,8,7],[1,9,5],[0,2]], [6,1,8,2])
Allocating PartitionStack...
Allocation passed:
(0 1 2 3 4 5 6 7 8 9)
Checking that entries are in order and correct level.
Everything seems in order, deallocating.
Deallocated.
Creating PartitionStack from partition [[6], [3, 4, 8, 7], [1, 9, 5], [0, 2]].
PartitionStack's data:
entries -> [6, 3, 4, 8, 7, 1, 9, 5, 0, 2]
levels -> [0, 10, 10, 10, 0, 10, 10, 0, 10, -1]
depth = 0, degree = 10
(6|3 4 8 7|1 9 5|0 2)
Checking PS_is_discrete:
False
Checking PS_num_cells:
4
Checking PS_is_mcr, min cell reps are:
[6, 3, 1, 0]
Checking PS_is_fixed, fixed elements are:
[6]
Copying PartitionStack:
(6|3 4 8 7|1 9 5|0 2)
Checking for consistency.
Everything is consistent.
Clearing copy:
(0 3 4 8 7 1 9 5 6 2)
Splitting point 6 from original:
0
(6|3 4 8 7|1 9 5|0 2)
Splitting point 1 from original:
5
(6|3 4 8 7|1|5 9|0 2)
Splitting point 8 from original:
1
(6|8|3 4 7|1|5 9|0 2)
Splitting point 2 from original:
8
(6|8|3 4 7|1|5 9|2|0)
Getting permutation from PS2->PS:
[6, 1, 0, 8, 3, 9, 2, 7, 4, 5]
Finding first smallest:
Minimal element is 5, bitset is:
0000010001
Finding element 1:
Location is: 5
Bitset is:
0100000000
Deallocating PartitionStacks.
Done.
```

sage.sets.disjoint_set.**SC_test_list_perms**(*L*, *n*, *limit*, *gap*, *limit_complain*, *test_contains*)

Test that the permutation group generated by list perms in L of degree n is of the correct order, by comparing with GAP. Don't test if the group is of size greater than limit.

TESTS:

```
sage: from sage.groups.perm_gps.partn_ref.automorphism_group_canonical_label import SC_test_list
sage: limit = 10^7
sage: def test_Sn_on_m_points(n, m, gap, contains):
...     perm1 = [1,0] + range(m)[2:]
...     perm2 = [(i+1)%n for i in range( n )] + range(m)[n:]
...     SC_test_list_perms([perm1, perm2], m, limit, gap, 0, contains)
sage: for i in range(2,9):
...     test_Sn_on_m_points(i,i,1,0)
sage: for i in range(2,9):
...     test_Sn_on_m_points(i,i,0,1)
sage: for i in range(2,9):           # long time
...     test_Sn_on_m_points(i,i,1,1) # long time
sage: test_Sn_on_m_points(8,8,1,1)
sage: def test_stab_chain_fns_1(n, gap, contains):
...     perm1 = sum([[2*i+1,2*i] for i in range(n)], [])
...     perm2 = [(i+1)%(2*n) for i in range( 2*n)]
...     SC_test_list_perms([perm1, perm2], 2*n, limit, gap, 0, contains)
sage: for n in range(1,11):
...     test_stab_chain_fns_1(n, 1, 0)
sage: for n in range(1,11):
...     test_stab_chain_fns_1(n, 0, 1)
sage: for n in range(1,9):              # long time
...     test_stab_chain_fns_1(n, 1, 1)  # long time
sage: test_stab_chain_fns_1(11, 1, 1)
sage: def test_stab_chain_fns_2(n, gap, contains):
...     perms = []
...     for p,e in factor(n):
...         perm1 = [(p*(i//p)) + ((i+1)%p) for i in range(n)]
...         perms.append(perm1)
...     SC_test_list_perms(perms, n, limit, gap, 0, contains)
sage: for n in range(2,11):
...     test_stab_chain_fns_2(n, 1, 0)
sage: for n in range(2,11):
...     test_stab_chain_fns_2(n, 0, 1)
sage: for n in range(2,11):             # long time
...     test_stab_chain_fns_2(n, 1, 1) # long time
sage: test_stab_chain_fns_2(11, 1, 1)
sage: def test_stab_chain_fns_3(n, gap, contains):
...     perm1 = [(-i)%n for i in range( n )]
...     perm2 = [(i+1)%n for i in range( n )]
...     SC_test_list_perms([perm1, perm2], n, limit, gap, 0, contains)
sage: for n in range(2,20):
...     test_stab_chain_fns_3(n, 1, 0)
sage: for n in range(2,20):
...     test_stab_chain_fns_3(n, 0, 1)
sage: for n in range(2,14):             # long time
...     test_stab_chain_fns_3(n, 1, 1) # long time
sage: test_stab_chain_fns_3(20, 1, 1)
sage: def test_stab_chain_fns_4(n, g, gap, contains):
...     perms = []
...     for _ in range(g):
...         perm = range(n)
...         shuffle(perm)
...         perms.append(perm)
...     SC_test_list_perms(perms, n, limit, gap, 0, contains)
```

```
sage: for n in range(4,9):                    # long time
...     test_stab_chain_fns_4(n, 1, 1, 0) # long time
...     test_stab_chain_fns_4(n, 2, 1, 0) # long time
...     test_stab_chain_fns_4(n, 2, 1, 0) # long time
...     test_stab_chain_fns_4(n, 2, 1, 0) # long time
...     test_stab_chain_fns_4(n, 2, 1, 0) # long time
...     test_stab_chain_fns_4(n, 3, 1, 0) # long time
sage: for n in range(4,9):
...     test_stab_chain_fns_4(n, 1, 0, 1)
...     for j in range(6):
...         test_stab_chain_fns_4(n, 2, 0, 1)
...     test_stab_chain_fns_4(n, 3, 0, 1)
sage: for n in range(4,8):                    # long time
...     test_stab_chain_fns_4(n, 1, 1, 1) # long time
...     test_stab_chain_fns_4(n, 2, 1, 1) # long time
...     test_stab_chain_fns_4(n, 2, 1, 1) # long time
...     test_stab_chain_fns_4(n, 3, 1, 1) # long time
sage: test_stab_chain_fns_4(8, 2, 1, 1)
sage: def test_stab_chain_fns_5(n, gap, contains):
...     perms = []
...     m = n//3
...     perm1 = range(2*m)
...     shuffle(perm1)
...     perm1 += range(2*m,n)
...     perm2 = range(m,n)
...     shuffle(perm2)
...     perm2 = range(m) + perm2
...     SC_test_list_perms([perm1, perm2], n, limit, gap, 0, contains)
sage: for n in [4..9]:                         # long time
...     for _ in range(2):                     # long time
...         test_stab_chain_fns_5(n, 1, 0) # long time
sage: for n in [4..8]:                         # long time
...     test_stab_chain_fns_5(n, 0, 1)         # long time
sage: for n in [4..9]:                         # long time
...     test_stab_chain_fns_5(n, 1, 1)         # long time
sage: def random_perm(x):
...     shuffle(x)
...     return x
sage: def test_stab_chain_fns_6(m,n,k, gap, contains):
...     perms = []
...     for i in range(k):
...         perm = sum([random_perm(range(i*(n//m),min(n,(i+1)*(n//m)))) for i in range(m)], [])
...         perms.append(perm)
...     SC_test_list_perms(perms, m*(n//m), limit, gap, 0, contains)
sage: for m in range(2,9):                             # long time
...     for n in range(m,3*m):                         # long time
...         for k in range(1,3):                       # long time
...             test_stab_chain_fns_6(m,n,k, 1, 0) # long time
sage: for m in range(2,10):
...     for n in range(m,4*m):
...         for k in range(1,3):
...             test_stab_chain_fns_6(m,n,k, 0, 1)
sage: test_stab_chain_fns_6(10,20,2, 1, 1)
sage: test_stab_chain_fns_6(8,16,2, 1, 1)
sage: test_stab_chain_fns_6(6,36,2, 1, 1)
sage: test_stab_chain_fns_6(4,40,3, 1, 1)
sage: test_stab_chain_fns_6(4,40,2, 1, 1)
sage: def test_stab_chain_fns_7(n, cop, gap, contains):
```

```
...     perms = []
...     for i in range(0,n//2,2):
...         p = range(n)
...         p[i] = i+1
...         p[i+1] = i
...     if cop:
...         perms.append([c for c in p])
...     else:
...         perms.append(p)
...     SC_test_list_perms(perms, n, limit, gap, 0, contains)
sage: for n in [6..14]:
...     test_stab_chain_fns_7(n, 1, 1, 0)
...     test_stab_chain_fns_7(n, 0, 1, 0)
sage: for n in [6..30]:
...     test_stab_chain_fns_7(n, 1, 0, 1)
...     test_stab_chain_fns_7(n, 0, 0, 1)
sage: for n in [6..14]:                        # long time
...     test_stab_chain_fns_7(n, 1, 1, 1) # long time
...     test_stab_chain_fns_7(n, 0, 1, 1) # long time
sage: test_stab_chain_fns_7(20, 1, 1, 1)
sage: test_stab_chain_fns_7(20, 0, 1, 1)
```

CHAPTER

TWENTYFOUR

DISJOINT UNION OF ENUMERATED SETS

AUTHORS:

- Florent Hivert (2009-07/09): initial implementation.
- Florent Hivert (2010-03): classcall related stuff.
- Florent Hivert (2010-12): fixed facade element construction.

class sage.sets.disjoint_union_enumerated_sets.**DisjointUnionEnumeratedSets**(*family*, *facade=True*, *keepkey=False*, *category=None*)

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

A class for disjoint unions of enumerated sets.

INPUT:

- family – a list (or iterable or family) of enumerated sets
- keepkey – a boolean
- facade – a boolean

This models the enumerated set obtained by concatenating together the specified ordered sets. The later are supposed to be pairwise disjoint; otherwise, a multiset is created.

The argument family can be a list, a tuple, a dictionary, or a family. If it is not a family it is first converted into a family (see sage.sets.family.Family()).

Experimental options:

By default, there is no way to tell from which set of the union an element is generated. The option keepkey=True keeps track of those by returning pairs (key, el) where key is the index of the set to which el belongs. When this option is specified, the enumerated sets need not be disjoint anymore.

With the option facade=False the elements are wrapped in an object whose parent is the disjoint union itself. The wrapped object can then be recovered using the 'value' attribute.

The two options can be combined.

The names of those options is imperfect, and subject to change in future versions. Feedback welcome.

EXAMPLES:

The input can be a list or a tuple of FiniteEnumeratedSets:

```
sage: U1 = DisjointUnionEnumeratedSets((
...         FiniteEnumeratedSet([1,2,3]),
...         FiniteEnumeratedSet([4,5,6])))
sage: U1
Disjoint union of Family ({1, 2, 3}, {4, 5, 6})
sage: U1.list()
[1, 2, 3, 4, 5, 6]
sage: U1.cardinality()
6
```

The input can also be a dictionary:

```
sage: U2 = DisjointUnionEnumeratedSets({1: FiniteEnumeratedSet([1,2,3]),
...                                     2: FiniteEnumeratedSet([4,5,6])})
sage: U2
Disjoint union of Finite family {1: {1, 2, 3}, 2: {4, 5, 6}}
sage: U2.list()
[1, 2, 3, 4, 5, 6]
sage: U2.cardinality()
6
```

However in that case the enumeration order is not specified.

In general the input can be any family:

```
sage: U3 = DisjointUnionEnumeratedSets(
...        Family([2,3,4], Permutations, lazy=True))
sage: U3
Disjoint union of Lazy family (<class 'sage.combinat.permutation.Permutations'>(i))_{i in [2, 3,
sage: U3.cardinality()
32
sage: it = iter(U3)
sage: [next(it), next(it), next(it), next(it), next(it), next(it)]
[[1, 2], [2, 1], [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1]]
sage: U3.unrank(18)
[2, 4, 1, 3]
```

This allows for infinite unions:

```
sage: U4 = DisjointUnionEnumeratedSets(
...        Family(NonNegativeIntegers(), Permutations))
sage: U4
Disjoint union of Lazy family (<class 'sage.combinat.permutation.Permutations'>(i))_{i in Non ne
sage: U4.cardinality()
+Infinity
sage: it = iter(U4)
sage: [next(it), next(it), next(it), next(it), next(it), next(it)]
[[], [1], [1, 2], [2, 1], [1, 2, 3], [1, 3, 2]]
sage: U4.unrank(18)
[2, 3, 1, 4]
```

Warning: Beware that some of the operations assume in that case that infinitely many of the enumerated sets are non empty.

Experimental options

We demonstrate the `keepkey` option:

```
sage: Ukeep = DisjointUnionEnumeratedSets(
...               Family(range(4), Permutations), keepkey=True)
sage: it = iter(Ukeep)
sage: [next(it) for i in range(6)]
[(0, []), (1, [1]), (2, [1, 2]), (2, [2, 1]), (3, [1, 2, 3]), (3, [1, 3, 2])]
sage: type(next(it)[1])
<class 'sage.combinat.permutation.StandardPermutations_n_with_category.element_class'>
```

We now demonstrate the `facade` option:

```
sage: UNoFacade = DisjointUnionEnumeratedSets(
...                   Family(range(4), Permutations), facade=False)
sage: it = iter(UNoFacade)
sage: [next(it) for i in range(6)]
[[], [1], [1, 2], [2, 1], [1, 2, 3], [1, 3, 2]]
sage: el = next(it); el
[2, 1, 3]
sage: type(el)
<type 'sage.structure.element_wrapper.ElementWrapper'>
sage: el.parent() == UNoFacade
True
sage: elv = el.value; elv
[2, 1, 3]
sage: type(elv)
<class 'sage.combinat.permutation.StandardPermutations_n_with_category.element_class'>
```

Possible extensions: the current enumeration order is not suitable for unions of infinite enumerated sets (except possibly for the last one). One could add options to specify alternative enumeration orders (anti-diagonal, round robin, ...) to handle this case.

Inheriting from `DisjointUnionEnumeratedSets`

There are two different use cases for inheriting from `DisjointUnionEnumeratedSets`: writing a parent which happens to be a disjoint union of some known parents, or writing generic disjoint unions for some particular classes of `sage.categories.enumerated_sets.EnumeratedSets`.

- In the first use case, the input of the `__init__` method is most likely different from that of `DisjointUnionEnumeratedSets`. Then, one simply writes the `__init__` method as usual:

```
sage: class MyUnion(DisjointUnionEnumeratedSets):
...     def __init__(self):
...         DisjointUnionEnumeratedSets.__init__(self,
...               Family([1,2], Permutations))
sage: pp = MyUnion()
sage: pp.list()
[[1], [1, 2], [2, 1]]
```

In case the `__init__()` method takes optional arguments, or does some normalization on them, a specific method `__classcall_private__` is required (see the documentation of `UniqueRepresentation`).

- In the second use case, the input of the `__init__` method is the same as that of `DisjointUnionEnumeratedSets`; one therefore wants to inherit the `__classcall_private__()` method as well, which can be achieved as follows:

```
sage: class UnionOfSpecialSets(DisjointUnionEnumeratedSets):
...     __classcall_private__ = staticmethod(DisjointUnionEnumeratedSets.__classcall_private_
...
sage: psp = UnionOfSpecialSets(Family([1,2], Permutations))
sage: psp.list()
[[1], [1, 2], [2, 1]]
```

TESTS:

```
sage: TestSuite(U1).run()
sage: TestSuite(U2).run()
sage: TestSuite(U3).run()
sage: TestSuite(U4).run()
doctest:...: UserWarning: Disjoint union of Lazy family (<class 'sage.combinat.permutation.Permu
The default implementation of __contains__ can loop forever. Please overload it.
sage: TestSuite(Ukeep).run()
sage: TestSuite(UNoFacade).run()
```

The following three lines are required for the pickling tests, because the classes `MyUnion` and `UnionOfSpecialSets` have been defined interactively:

```
sage: import __main__
sage: __main__.MyUnion = MyUnion
sage: __main__.UnionOfSpecialSets = UnionOfSpecialSets

sage: TestSuite(pp).run()
sage: TestSuite(psp).run()
```

Element()

TESTS:

```
sage: U = DisjointUnionEnumeratedSets(
...           Family([1,2,3], Partitions), facade=False)
sage: U.Element
<type 'sage.structure.element_wrapper.ElementWrapper'>
sage: U = DisjointUnionEnumeratedSets(
...           Family([1,2,3], Partitions), facade=True)
sage: U.Element
Traceback (most recent call last):
...
AttributeError: 'DisjointUnionEnumeratedSets_with_category' object has no attribute 'Element
```

an_element()

Returns an element of this disjoint union, as per `Sets.ParentMethods.an_element()`.

EXAMPLES:

```
sage: U4 = DisjointUnionEnumeratedSets(
...           Family([3, 5, 7], Permutations))
sage: U4.an_element()
[1, 2, 3]
```

cardinality()

Returns the cardinality of this disjoint union.

EXAMPLES:

For finite disjoint unions, the cardinality is computed by summing the cardinalities of the enumerated sets:

```
sage: U = DisjointUnionEnumeratedSets(Family([0,1,2,3], Permutations))
sage: U.cardinality()
10
```

For infinite disjoint unions, this makes the assumption that the result is infinite:

```
sage: U = DisjointUnionEnumeratedSets(
...           Family(NonNegativeIntegers(), Permutations))
sage: U.cardinality()
+Infinity
```

Warning: as pointed out in the main documentation, it is possible to construct examples where this is incorrect:

```
sage: U = DisjointUnionEnumeratedSets(
...           Family(NonNegativeIntegers(), lambda x: []))
sage: U.cardinality()  # Should be 0!
+Infinity
```

CHAPTER

TWENTYFIVE

ENUMERATED SET FROM ITERATOR

EXAMPLES:

We build a set from the iterator `graphs` that returns a canonical representative for each isomorphism class of graphs:

```
sage: from sage.sets.set_from_iterator import EnumeratedSetFromIterator
sage: E = EnumeratedSetFromIterator(
...     graphs,
...     name = "Graphs",
...     category = InfiniteEnumeratedSets(),
...     cache = True)
sage: E
Graphs
sage: E.unrank(0)
Graph on 0 vertices
sage: E.unrank(4)
Graph on 3 vertices
sage: E.cardinality()
+Infinity
sage: E.category()
Category of facade infinite enumerated sets
```

The module also provides decorator for functions and methods:

```
sage: from sage.sets.set_from_iterator import set_from_function
sage: @set_from_function
... def f(n): return xsrange(n)
sage: f(3)
{0, 1, 2}
sage: f(5)
{0, 1, 2, 3, 4}
sage: f(100)
{0, 1, 2, 3, 4, ...}
```

```
sage: from sage.sets.set_from_iterator import set_from_method
sage: class A:
...    @set_from_method
...    def f(self,n):
...        return xsrange(n)
sage: a = A()
sage: a.f(3)
{0, 1, 2}
sage: f(100)
{0, 1, 2, 3, 4, ...}
```

class sage.sets.set_from_iterator.**Decorator**

Abstract class that manage documentation and sources of the wrapped object.

The method needs to be stored in the attribute self.f

class sage.sets.set_from_iterator.**DummyExampleForPicklingTest**

Class example to test pickling with the decorator set_from_method.

> **Warning:** This class is intended to be used in doctest only.

EXAMPLES:

```
sage: from sage.sets.set_from_iterator import DummyExampleForPicklingTest
sage: DummyExampleForPicklingTest().f()
{10, 11, 12, 13, 14, ...}
```

f()

Returns the set between self.start and self.stop.

EXAMPLES:

```
sage: from sage.sets.set_from_iterator import DummyExampleForPicklingTest
sage: d = DummyExampleForPicklingTest()
sage: d.f()
{10, 11, 12, 13, 14, ...}
sage: d.start = 4
sage: d.stop = 200
sage: d.f()
{4, 5, 6, 7, 8, ...}
```

class sage.sets.set_from_iterator.**EnumeratedSetFromIterator**(*f*, *args=None*, *kwds=None*, *name=None*, *category=None*, *cache=False*)

Bases: sage.structure.parent.Parent

A class for enumerated set built from an iterator.

INPUT:

- f – a function that returns an iterable from which the set is built from
- args – tuple – arguments to be sent to the function f
- kwds – dictionnary – keywords to be sent to the function f
- name – an optional name for the set
- category – (default: None) an optional category for that enumerated set. If you know that your iterator will stop after a finite number of steps you should set it as FiniteEnumeratedSets, conversly if you know that your iterator will run over and over you should set it as InfiniteEnumeratedSets.
- cache – boolean (default: False) – Whether or not use a cache mechanism for the iterator. If True, then the function f is called only once.

EXAMPLES:

```
sage: from sage.sets.set_from_iterator import EnumeratedSetFromIterator
sage: E = EnumeratedSetFromIterator(graphs, args = (7,))
sage: E
{Graph on 7 vertices, Graph on 7 vertices, Graph on 7 vertices, Graph on 7 vertices, Graph on 7
```

```
sage: E.category()
Category of facade enumerated sets
```

The same example with a cache and a custom name:

```
sage: E = EnumeratedSetFromIterator(
...       graphs,
...       args = (8,),
...       category = FiniteEnumeratedSets(),
...       name = "Graphs with 8 vertices",
...       cache = True)
sage: E
Graphs with 8 vertices
sage: E.unrank(3)
Graph on 8 vertices
sage: E.category()
Category of facade finite enumerated sets
```

TESTS:

The cache is compatible with multiple call to `__iter__`:

```
sage: from itertools import count
sage: E = EnumeratedSetFromIterator(count, args=(0,), category=InfiniteEnumeratedSets(), cache=T
sage: e1 = iter(E)
sage: e2 = iter(E)
sage: next(e1), next(e1)
(0, 1)
sage: next(e2), next(e2), next(e2)
(0, 1, 2)
sage: next(e1), next(e1)
(2, 3)
sage: next(e2)
3
```

The following warning is due to `E` being a facade parent. For more, see the discussion on trac ticket #16239:

```
sage: TestSuite(E).run()
doctest:...: UserWarning: Testing equality of infinite sets which will not end in case of equali

sage: E = EnumeratedSetFromIterator(xsrange, args=(10,), category=FiniteEnumeratedSets(), cache=
sage: TestSuite(E).run()
```

Note: In order to make the `TestSuite` works, the elements of the set should have parents.

clear_cache()

Clear the cache.

EXAMPLES:

```
sage: from itertools import count
sage: from sage.sets.set_from_iterator import EnumeratedSetFromIterator
sage: E = EnumeratedSetFromIterator(count, args=(1,), cache=True)
sage: e1 = E._cache
sage: e1
lazy list [1, 2, 3, ...]
sage: E.clear_cache()
sage: E._cache
lazy list [1, 2, 3, ...]
```

```
sage: e1 is E._cache
False
```

is_parent_of(*x*)

Test whether `x` is in `self`.

If the set is infinite, only the answer `True` should be expected in finite time.

EXAMPLES:

```
sage: from sage.sets.set_from_iterator import EnumeratedSetFromIterator
sage: P = Partitions(12,min_part=2,max_part=5)
sage: E = EnumeratedSetFromIterator(P.__iter__)
sage: P([5,5,2]) in E
True
```

unrank(*i*)

Returns the element at position `i`.

EXAMPLES:

```
sage: from sage.sets.set_from_iterator import EnumeratedSetFromIterator
sage: E = EnumeratedSetFromIterator(graphs, args=(8,), cache=True)
sage: F = EnumeratedSetFromIterator(graphs, args=(8,), cache=False)
sage: E.unrank(2)
Graph on 8 vertices
sage: E.unrank(2) == F.unrank(2)
True
```

class `sage.sets.set_from_iterator.`**EnumeratedSetFromIterator_function_decorator**(*f=None*, *name=None*, ***options*)

Bases: `sage.sets.set_from_iterator.Decorator`

Decorator for `EnumeratedSetFromIterator`.

Name could be string or a function `(args,kwds) -> string`.

> **Warning:** If you are going to use this with the decorator `cached_function`, you must place the `cached_function` first. See the example below.

EXAMPLES:

```
sage: from sage.sets.set_from_iterator import set_from_function
sage: @set_from_function
... def f(n):
...     for i in xrange(n):
...         yield i**2 + i + 1
sage: f(3)
{1, 3, 7}
sage: f(100)
{1, 3, 7, 13, 21, ...}
```

To avoid ambiguity, it is always better to use it with a call which provides optional global initialization for the call to `EnumeratedSetFromIterator`:

```
sage: @set_from_function(category=InfiniteEnumeratedSets())
... def Fibonacci():
...     a = 1; b = 2
```

```
...     while True:
...         yield a
...         a,b = b,a+b
sage: F = Fibonacci()
sage: F
{1, 2, 3, 5, 8, ...}
sage: F.cardinality()
+Infinity
```

A simple example with many options:

```
sage: @set_from_function(
...         name = "From %(m)d to %(n)d",
...         category = FiniteEnumeratedSets())
... def f(m,n): return xsrange(m,n+1)
sage: E = f(3,10); E
From 3 to 10
sage: E.list()
[3, 4, 5, 6, 7, 8, 9, 10]
sage: E = f(1,100); E
From 1 to 100
sage: E.cardinality()
100
sage: f(n=100,m=1) == E
True
```

An example which mixes together `set_from_function` and `cached_method`:

```
sage: @cached_function
... @set_from_function(
...     name = "Graphs on %(n)d vertices",
...     category = FiniteEnumeratedSets(),
...     cache = True)
... def Graphs(n): return graphs(n)
sage: Graphs(10)
Graphs on 10 vertices
sage: Graphs(10).unrank(0)
Graph on 10 vertices
sage: Graphs(10) is Graphs(10)
True
```

The `cached_function` must go first:

```
sage: @set_from_function(
...     name = "Graphs on %(n)d vertices",
...     category = FiniteEnumeratedSets(),
...     cache = True)
... @cached_function
... def Graphs(n): return graphs(n)
sage: Graphs(10)
Graphs on 10 vertices
sage: Graphs(10).unrank(0)
Graph on 10 vertices
sage: Graphs(10) is Graphs(10)
False
```

class sage.sets.set_from_iterator.**EnumeratedSetFromIterator_method_caller**(*inst*, *f*, *name=None*, ***options*)

Bases: sage.sets.set_from_iterator.Decorator

Caller for decorated method in class.

INPUT:

- inst – an instance of a class
- f – a method of a class of inst (and not of the instance itself)
- name – optional – either a string (which may contains substitution rules from argument or a function args,kwds -> string.
- options – any option accepted by EnumeratedSetFromIterator

class sage.sets.set_from_iterator.**EnumeratedSetFromIterator_method_decorator**(*f=None*, ***options*)

Bases: object

Decorator for enumerated set built from a method.

INPUT:

- f – Optional function from which are built the enumerated sets at each call
- name – Optional string (which may contains substitution rules from argument) or a function (args,kwds) -> string.
- any option accepted by EnumeratedSetFromIterator.

EXAMPLES:

```
sage: from sage.sets.set_from_iterator import set_from_method
sage: class A():
...     def n(self): return 12
...     @set_from_method
...     def f(self): return xsrange(self.n())
sage: a = A()
sage: print a.f.__class__
sage.sets.set_from_iterator.EnumeratedSetFromIterator_method_caller
sage: a.f()
{0, 1, 2, 3, 4, ...}
sage: A.f(a)
{0, 1, 2, 3, 4, ...}
```

A more complicated example with a parametrized name:

```
sage: class B():
...     @set_from_method(
...         name = "Graphs(%(n)d)",
...         category = FiniteEnumeratedSets())
...     def graphs(self, n): return graphs(n)
sage: b = B()
sage: G3 = b.graphs(3)
sage: G3
Graphs(3)
sage: G3.cardinality()
4
```

```
sage: G3.category()
Category of facade finite enumerated sets
sage: B.graphs(b,3)
Graphs(3)
```

And a last example with a name parametrized by a function:

```
sage: class D():
...     def __init__(self, name): self.name = str(name)
...     def __str__(self): return self.name
...     @set_from_method(
...         name = lambda self,n: str(self)*n,
...         category = FiniteEnumeratedSets())
...     def subset(self, n):
...         return xsrange(n)
sage: d = D('a')
sage: E = d.subset(3); E
aaa
sage: E.list()
[0, 1, 2]
sage: F = d.subset(n=10); F
aaaaaaaaaa
sage: F.list()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Todo

It is not yet possible to use `set_from_method` in conjunction with `cached_method`.

`sage.sets.set_from_iterator.`**`set_from_function`**
alias of `EnumeratedSetFromIterator_function_decorator`

`sage.sets.set_from_iterator.`**`set_from_method`**
alias of `EnumeratedSetFromIterator_method_decorator`

CHAPTER

TWENTYSIX

FINITE ENUMERATED SETS

class sage.sets.finite_enumerated_set.**FiniteEnumeratedSet**(*elements*)

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

A class for finite enumerated set.

Returns the finite enumerated set with elements in elements where element is any (finite) iterable object.

The main purpose is to provide a variant of list or tuple, which is a parent with an interface consistent with EnumeratedSets and has unique representation. The list of the elements is expanded in memory.

EXAMPLES:

```
sage: S = FiniteEnumeratedSet([1, 2, 3])
sage: S
{1, 2, 3}
sage: S.list()
[1, 2, 3]
sage: S.cardinality()
3
sage: S.random_element()
1
sage: S.first()
1
sage: S.category()
Category of facade finite enumerated sets
sage: TestSuite(S).run()
```

Note that being and enumerated set, the result depends on the order:

```
sage: S1 = FiniteEnumeratedSet((1, 2, 3))
sage: S1
{1, 2, 3}
sage: S1.list()
[1, 2, 3]
sage: S1 == S
True
sage: S2 = FiniteEnumeratedSet((2, 1, 3))
sage: S2 == S
False
```

As an abuse, repeated entries in elements are allowed to model multisets:

```
sage: S1 = FiniteEnumeratedSet((1, 2, 1, 2, 2, 3))
sage: S1
{1, 2, 1, 2, 2, 3}
```

Finaly the elements are not aware of their parent:

```
sage: S.first().parent()
Integer Ring
```

an_element()

TESTS:

```
sage: S = FiniteEnumeratedSet([1,2,3])
sage: S.an_element()
1
```

cardinality()

TESTS:

```
sage: S = FiniteEnumeratedSet([1,2,3])
sage: S.cardinality()
3
```

first()

Return the first element of the enumeration or raise an EmptySetError if the set is empty.

EXAMPLES:

```
sage: S = FiniteEnumeratedSet('abc')
sage: S.first()
'a'
```

index(*x*)

Returns the index of x in this finite enumerated set.

EXAMPLES:

```
sage: S = FiniteEnumeratedSet(['a','b','c'])
sage: S.index('b')
1
```

is_parent_of(*x*)

TESTS:

```
sage: S = FiniteEnumeratedSet([1,2,3])
sage: 1 in S
True
sage: 2 in S
True
sage: 4 in S
False
sage: ZZ in S
False

sage: S.is_parent_of(2)
True
sage: S.is_parent_of(4)
False
```

last()

Returns the last element of the iteration or raise an EmptySetError if the set is empty.

EXAMPLES:

```
sage: S = FiniteEnumeratedSet([0,'a',1.23, 'd'])
sage: S.last()
'd'
```

list()

TESTS:

```
sage: S = FiniteEnumeratedSet([1,2,3])
sage: S.list()
[1, 2, 3]
```

random_element()

Return a random element.

EXAMPLES:

```
sage: S = FiniteEnumeratedSet('abc')
sage: S.random_element()    # random
'b'
```

rank(*x*)

Returns the index of x in this finite enumerated set.

EXAMPLES:

```
sage: S = FiniteEnumeratedSet(['a','b','c'])
sage: S.index('b')
1
```

unrank(*i*)

Return the element at position i.

EXAMPLES:

```
sage: S = FiniteEnumeratedSet([1,'a',-51])
sage: S[0], S[1], S[2]
(1, 'a', -51)
sage: S[3]
Traceback (most recent call last):
...
IndexError: list index out of range
sage: S[-1], S[-2], S[-3]
(-51, 'a', 1)
sage: S[-4]
Traceback (most recent call last):
...
IndexError: list index out of range
```

CHAPTER

TWENTYSEVEN

RECURSIVELY ENUMERATED SET

A set S is called recursively enumerable if there is an algorithm that enumerates the members of S. We consider here the recursively enumerated sets that are described by some `seeds` and a successor function `successors`. The successor function may have some structure (symmetric, graded, forest) or not. The elements of a set having a symmetric, graded or forest structure can be enumerated uniquely without keeping all of them in memory. Many kinds of iterators are provided in this module: depth first search, breadth first search or elements of given depth.

See Wikipedia article Recursively_enumerable_set.

See documentation of `RecursivelyEnumeratedSet()` below for the description of the inputs.

AUTHORS:

- Sebastien Labbe, April 2014, at Sage Days 57, Cernay-la-ville

EXAMPLES:

27.1 Forest structure

The set of words over the alphabet $\{a, b\}$ can be generated from the empty word by appending letter a or b as a successor function. This set has a forest structure:

```
sage: seeds = ['']
sage: succ = lambda w: [w+'a', w+'b']
sage: C = RecursivelyEnumeratedSet(seeds, succ, structure='forest')
sage: C
An enumerated set with a forest structure
```

Depth first search iterator:

```
sage: it = C.depth_first_search_iterator()
sage: [next(it) for _ in range(6)]
['', 'a', 'aa', 'aaa', 'aaaa', 'aaaaa']
```

Breadth first search iterator:

```
sage: it = C.breadth_first_search_iterator()
sage: [next(it) for _ in range(6)]
['', 'a', 'b', 'aa', 'ab', 'ba']
```

27.2 Symmetric structure

The origin `(0, 0)` as seed and the upper, lower, left and right lattice point as successor function. This function is symmetric:

```
sage: succ = lambda a: [(a[0]-1,a[1]), (a[0],a[1]-1), (a[0]+1,a[1]), (a[0],a[1]+1)]
sage: seeds = [(0,0)]
sage: C = RecursivelyEnumeratedSet(seeds, succ, structure='symmetric', enumeration='depth')
sage: C
A recursively enumerated set with a symmetric structure (depth first search)
```

In this case, depth first search is the default enumeration for iteration:

```
sage: it_depth = iter(C)
sage: [next(it_depth) for _ in range(10)]
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9)]
```

Breadth first search:

```
sage: it_breadth = C.breadth_first_search_iterator()
sage: [next(it_breadth) for _ in range(10)]
[(0, 0), (0, 1), (0, -1), (1, 0), (-1, 0), (-1, 1), (-2, 0), (0, 2), (2, 0), (-1, -1)]
```

Levels (elements of given depth):

```
sage: sorted(C.graded_component(0))
[(0, 0)]
sage: sorted(C.graded_component(1))
[(-1, 0), (0, -1), (0, 1), (1, 0)]
sage: sorted(C.graded_component(2))
[(-2, 0), (-1, -1), (-1, 1), (0, -2), (0, 2), (1, -1), (1, 1), (2, 0)]
```

27.3 Graded structure

Identity permutation as seed and `permutohedron_succ` as successor function:

```
sage: succ = attrcall("permutohedron_succ")
sage: seed = [Permutation([1..5])]
sage: R = RecursivelyEnumeratedSet(seed, succ, structure='graded')
sage: R
A recursively enumerated set with a graded structure (breadth first search)
```

Depth first search iterator:

```
sage: it_depth = R.depth_first_search_iterator()
sage: [next(it_depth) for _ in range(5)]
[[1, 2, 3, 4, 5],
 [1, 2, 3, 5, 4],
 [1, 2, 5, 3, 4],
 [1, 2, 5, 4, 3],
 [1, 5, 2, 4, 3]]
```

Breadth first search iterator:

```
sage: it_breadth = R.breadth_first_search_iterator()
sage: [next(it_breadth) for _ in range(5)]
[[1, 2, 3, 4, 5],
 [1, 3, 2, 4, 5],
 [1, 2, 4, 3, 5],
 [2, 1, 3, 4, 5],
 [1, 2, 3, 5, 4]]
```

Elements of given depth iterator:

```
sage: list(R.elements_of_depth_iterator(9))
[[5, 3, 4, 2, 1], [4, 5, 3, 2, 1], [5, 4, 2, 3, 1], [5, 4, 3, 1, 2]]
sage: list(R.elements_of_depth_iterator(10))
[[5, 4, 3, 2, 1]]
```

Graded components (set of elements of the same depth):

```
sage: sorted(R.graded_component(0))
[[1, 2, 3, 4, 5]]
sage: sorted(R.graded_component(1))
[[1, 2, 3, 5, 4], [1, 2, 4, 3, 5], [1, 3, 2, 4, 5], [2, 1, 3, 4, 5]]
sage: sorted(R.graded_component(9))
[[4, 5, 3, 2, 1], [5, 3, 4, 2, 1], [5, 4, 2, 3, 1], [5, 4, 3, 1, 2]]
sage: sorted(R.graded_component(10))
[[5, 4, 3, 2, 1]]
```

27.4 No hypothesis on the structure

By "no hypothesis" is meant neither a forest, neither symmetric neither graded, it may have other structure like not containing oriented cycle but this does not help for enumeration.

In this example, the seed is 0 and the successor function is either `+2` or `+3`. This is the set of non negative linear combinations of 2 and 3:

```
sage: succ = lambda a:[a+2,a+3]
sage: C = RecursivelyEnumeratedSet([0], succ)
sage: C
A recursively enumerated set (breadth first search)
```

Breadth first search:

```
sage: it = C.breadth_first_search_iterator()
sage: [next(it) for _ in range(10)]
[0, 2, 3, 4, 5, 6, 8, 9, 7, 10]
```

Depth first search:

```
sage: it = C.depth_first_search_iterator()
sage: [next(it) for _ in range(10)]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

`sage.sets.recursively_enumerated_set.`**`RecursivelyEnumeratedSet`**`(`*seeds, successors, structure=None, enumeration=None, max_depth=None, post_process=None, facade=None, category=None*`)`

Return a recursively enumerated set.

A set S is called recursively enumerable if there is an algorithm that enumerates the members of S. We consider here the recursively enumerated set that are described by some `seeds` and a successor function `successors`.

Let U be a set and `successors` $: U \to 2^U$ be a successor function associating to each element of U a subset of U. Let `seeds` be a subset of U. Let $S \subseteq U$ be the set of elements of U that can be reached from a seed by applying recursively the `successors` function. This class provides different kinds of iterators (breadth first, depth first, elements of given depth, etc.) for the elements of S.

See Wikipedia article Recursively_enumerable_set.

INPUT:

- `seeds` – list (or iterable) of hashable objects
- `successors` – function (or callable) returning a list (or iterable) of hashable objects
- `structure` – string (optional, default: `None`), structure of the set, possible values are:
  - `None` – nothing is known about the structure of the set.
  - `'forest'` – if the `successors` function generates a *forest*, that is, each element can be reached uniquely from a seed.
  - `'graded'` – if the `successors` function is *graded*, that is, all paths from a seed to a given element have equal length.
  - `'symmetric'` – if the relation is *symmetric*, that is, `y in successors(x)` if and only if `x in successors(y)`
- `enumeration` – `'depth'`, `'breadth'`, `'naive'` or `None` (optional, default: `None`). The default enumeration for the `__iter__` function.
- `max_depth` – integer (optional, default: `float("inf")`), limit the search to a certain depth, currently works only for breadth first search
- `post_process` – (optional, default: `None`), for forest only
- `facade` – (optional, default: `None`)
- `category` – (optional, default: `None`)

EXAMPLES:

A recursive set with no other information:

```
sage: f = lambda a: [a+3, a+5]
sage: C = RecursivelyEnumeratedSet([0], f)
sage: C
A recursively enumerated set (breadth first search)
sage: it = iter(C)
sage: [next(it) for _ in range(10)]
[0, 3, 5, 8, 10, 6, 9, 11, 13, 15]
```

A recursive set with a forest structure:

```
sage: f = lambda a: [2*a,2*a+1]
sage: C = RecursivelyEnumeratedSet([1], f, structure='forest')
sage: C
An enumerated set with a forest structure
sage: it = C.depth_first_search_iterator()
sage: [next(it) for _ in range(7)]
[1, 2, 4, 8, 16, 32, 64]
sage: it = C.breadth_first_search_iterator()
sage: [next(it) for _ in range(7)]
[1, 2, 3, 4, 5, 6, 7]
```

A recursive set given by a symmetric relation:

```
sage: f = lambda a: [a-1,a+1]
sage: C = RecursivelyEnumeratedSet([10, 15], f, structure='symmetric')
sage: C
A recursively enumerated set with a symmetric structure (breadth first search)
sage: it = iter(C)
sage: [next(it) for _ in range(7)]
[10, 15, 16, 9, 11, 14, 8]
```

A recursive set given by a graded relation:

```
sage: f = lambda a: [a+1, a+I]
sage: C = RecursivelyEnumeratedSet([0], f, structure='graded')
sage: C
A recursively enumerated set with a graded structure (breadth first search)
sage: it = iter(C)
sage: [next(it) for _ in range(7)]
[0, 1, I, I + 1, 2, 2*I, I + 2]
```

> **Warning:** If you do not set the good structure, you might obtain bad results, like elements generated twice:
>
> ```
> sage: f = lambda a: [a-1,a+1]
> sage: C = RecursivelyEnumeratedSet([0], f, structure='graded')
> sage: it = iter(C)
> sage: [next(it) for _ in range(7)]
> [0, 1, -1, 0, 2, -2, 1]
> ```

TESTS:

The succesors method is an attribute:

```
sage: R = RecursivelyEnumeratedSet([1], lambda x: [x+1, x-1])
sage: R.successors(4)
[5, 3]
```

```
sage: C = RecursivelyEnumeratedSet((1, 2, 3), factor)
sage: C.successors
<function factor at ...>
sage: C._seeds
(1, 2, 3)
```

class sage.sets.recursively_enumerated_set.**RecursivelyEnumeratedSet_generic**

Bases: sage.structure.parent.Parent

A generic recursively enumerated set.

For more information, see `RecursivelyEnumeratedSet()`.

EXAMPLES:

```
sage: f = lambda a:[a+1]
```

Different structure for the sets:

```
sage: RecursivelyEnumeratedSet([0], f, structure=None)
A recursively enumerated set (breadth first search)
sage: RecursivelyEnumeratedSet([0], f, structure='graded')
A recursively enumerated set with a graded structure (breadth first search)
sage: RecursivelyEnumeratedSet([0], f, structure='symmetric')
A recursively enumerated set with a symmetric structure (breadth first search)
sage: RecursivelyEnumeratedSet([0], f, structure='forest')
An enumerated set with a forest structure
```

Different default enumeration algorithms:

```
sage: RecursivelyEnumeratedSet([0], f, enumeration='breadth')
A recursively enumerated set (breadth first search)
sage: RecursivelyEnumeratedSet([0], f, enumeration='naive')
A recursively enumerated set (naive search)
sage: RecursivelyEnumeratedSet([0], f, enumeration='depth')
A recursively enumerated set (depth first search)
```

breadth_first_search_iterator(*max_depth=None*)

Iterate on the elements of `self` (breadth first).

This code remembers every elements generated.

INPUT:

- `max_depth` – (Default: `None`) specifies the maximal depth to which elements are computed; if `None`, the value of `self._max_depth` is used

EXAMPLES:

```
sage: f = lambda a: [a+3, a+5]
sage: C = RecursivelyEnumeratedSet([0], f)
sage: it = C.breadth_first_search_iterator()
sage: [next(it) for _ in range(10)]
[0, 3, 5, 8, 10, 6, 9, 11, 13, 15]
```

depth_first_search_iterator()

Iterate on the elements of `self` (depth first).

This code remembers every elements generated.

See Wikipedia article Depth-first_search.

EXAMPLES:

```
sage: f = lambda a: [a+3, a+5]
sage: C = RecursivelyEnumeratedSet([0], f)
sage: it = C.depth_first_search_iterator()
sage: [next(it) for _ in range(10)]
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

elements_of_depth_iterator(*depth*)

Iterate over the elements of `self` of given depth.

An element of depth n can be obtained applying n times the successor function to a seed.

INPUT:

•`depth` – integer

OUTPUT:

An iterator.

EXAMPLES:

```
sage: f = lambda a: [a-1, a+1]
sage: S = RecursivelyEnumeratedSet([5, 10], f, structure='symmetric')
sage: it = S.elements_of_depth_iterator(2)
sage: sorted(it)
[3, 7, 8, 12]
```

graded_component (*depth*)

Return the graded component of given depth.

This method caches each lower graded component.

A graded component is a set of elements of the same depth where the depth of an element is its minimal distance to a root.

INPUT:

•`depth` – integer

OUTPUT:

A set.

EXAMPLES:

```
sage: f = lambda a: [a+3, a+5]
sage: C = RecursivelyEnumeratedSet([0], f)
sage: C.graded_component(0)
Traceback (most recent call last):
...
NotImplementedError: graded_component_iterator method currently implemented only for graded
```

When the structure is symmetric:

```
sage: f = lambda a: [a-1,a+1]
sage: C = RecursivelyEnumeratedSet([10, 15], f, structure='symmetric')
sage: for i in range(5): sorted(C.graded_component(i))
[10, 15]
[9, 11, 14, 16]
[8, 12, 13, 17]
[7, 18]
[6, 19]
```

When the structure is graded:

```
sage: f = lambda a: [a+1, a+I]
sage: C = RecursivelyEnumeratedSet([0], f, structure='graded')
sage: for i in range(5): sorted(C.graded_component(i))
[0]
[I, 1]
[2*I, I + 1, 2]
[3*I, 2*I + 1, I + 2, 3]
[4*I, 3*I + 1, 2*I + 2, I + 3, 4]
```

graded_component_iterator()

Iterate over the graded components of `self`.

A graded component is a set of elements of the same depth.

It is currently implemented only for herited classes.

OUTPUT:

An iterator of sets.

EXAMPLES:

```
sage: f = lambda a: [a+3, a+5]
sage: C = RecursivelyEnumeratedSet([0], f)
sage: it = C.graded_component_iterator()     # todo: not implemented
```

naive_search_iterator()

Iterate on the elements of `self` (in no particular order).

This code remembers every elements generated.

TESTS:

We compute all the permutations of 3:

```
sage: seeds = [Permutation([1,2,3])]
sage: succ = attrcall("permutohedron_succ")
sage: R = RecursivelyEnumeratedSet(seeds, succ)
sage: list(R.naive_search_iterator())
[[1, 2, 3], [2, 1, 3], [1, 3, 2], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

seeds()

Return an iterable over the seeds of `self`.

EXAMPLES:

```
sage: R = RecursivelyEnumeratedSet([1], lambda x: [x+1, x-1])
sage: R.seeds()
[1]
```

successors

class sage.sets.recursively_enumerated_set.**RecursivelyEnumeratedSet_graded**

Bases: `sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_generic`

Generic tool for constructing ideals of a graded relation.

INPUT:

- `seeds` – list (or iterable) of hashable objects
- `successors` – function (or callable) returning a list (or iterable)
- `enumeration` – `'depth'`, `'breadth'` or `None` (default: `None`)
- `max_depth` – integer (default: `float("inf")`)

EXAMPLES:

```
sage: f = lambda a: [(a[0]+1,a[1]), (a[0],a[1]+1)]
sage: C = RecursivelyEnumeratedSet([(0,0)], f, structure='graded', max_depth=3)
sage: C
A recursively enumerated set with a graded structure (breadth first search)
sage: sorted(C)
```

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

breadth_first_search_iterator (*max_depth=None*)

Iterate on the elements of `self` (breadth first).

This iterator make use of the graded structure by remembering only the elements of the current depth.

INPUT:

- `max_depth` – (Default: `None`) Specifies the maximal depth to which elements are computed. If None, the value of `self._max_depth` is used.

EXAMPLES:

```
sage: f = lambda a: [(a[0]+1,a[1]), (a[0],a[1]+1)]
sage: C = RecursivelyEnumeratedSet([(0,0)], f, structure='graded')
sage: it = C.breadth_first_search_iterator(max_depth=3)
sage: list(it)
[(0, 0), (0, 1), (1, 0), (2, 0), (1, 1),
 (0, 2), (3, 0), (1, 2), (0, 3), (2, 1)]
```

graded_component_iterator ()

Iterate over the graded components of `self`.

A graded component is a set of elements of the same depth.

The algorithm remembers only the current graded component generated since the structure is graded.

OUTPUT:

An iterator of sets.

EXAMPLES:

```
sage: f = lambda a: [(a[0]+1,a[1]), (a[0],a[1]+1)]
sage: C = RecursivelyEnumeratedSet([(0,0)], f, structure='graded', max_depth=3)
sage: it = C.graded_component_iterator()
sage: for _ in range(4): sorted(next(it))
[(0, 0)]
[(0, 1), (1, 0)]
[(0, 2), (1, 1), (2, 0)]
[(0, 3), (1, 2), (2, 1), (3, 0)]
```

class `sage.sets.recursively_enumerated_set.`**RecursivelyEnumeratedSet_symmetric**

Bases: `sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_generic`

Generic tool for constructing ideals of a symmetric relation.

INPUT:

- `seeds` – list (or iterable) of hashable objects
- `successors` – function (or callable) returning a list (or iterable)
- `enumeration` – `'depth'`, `'breadth'` or `None` (default: `None`)
- `max_depth` – integer (default: `float("inf")`)

EXAMPLES:

```
sage: f = lambda a: [a-1,a+1]
sage: C = RecursivelyEnumeratedSet([0], f, structure='symmetric')
sage: C
A recursively enumerated set with a symmetric structure (breadth first search)
```

```
sage: it = iter(C)
sage: [next(it) for _ in range(7)]
[0, 1, -1, 2, -2, 3, -3]
```

TESTS:

Do not use lambda functions for saving purposes:

```
sage: f = lambda a: [a-1,a+1]
sage: C = RecursivelyEnumeratedSet([0], f, structure='symmetric')
sage: loads(dumps(C))
Traceback (most recent call last):
...
PicklingError: Can't pickle <type 'function'>: attribute lookup __builtin__.function failed
```

This works in the command line but apparently not as a doctest:

```
sage: def f(a): return [a-1,a+1]
sage: C = RecursivelyEnumeratedSet([0], f, structure='symmetric')
sage: loads(dumps(C))
Traceback (most recent call last):
...
PicklingError: Can't pickle <type 'function'>: attribute lookup __builtin__.function failed
```

breadth_first_search_iterator(*max_depth=None*)

Iterate on the elements of `self` (breadth first).

INPUT:

- `max_depth` – (Default: `None`) specifies the maximal depth to which elements are computed; if `None`, the value of `self._max_depth` is used

---

Note: It should be slower than the other one since it must generates the whole graded component before yielding the first element of each graded component. It is used for test only.

EXAMPLES:

```
sage: f = lambda a: [(a[0]+1,a[1]), (a[0],a[1]+1)]
sage: C = RecursivelyEnumeratedSet([(0,0)], f, structure='graded')
sage: it = C._breadth_first_search_iterator_from_graded_component_iterator(max_depth=3)
sage: list(it)
[(0, 0), (0, 1), (1, 0), (2, 0), (1, 1), (0, 2)]
```

This iterator is used by default for symmetric structure:

```
sage: f = lambda a: [a-1,a+1]
sage: S = RecursivelyEnumeratedSet([10], f, structure='symmetric')
sage: it = iter(S)
sage: [next(it) for _ in range(7)]
[10, 9, 11, 8, 12, 13, 7]
```

graded_component_iterator()

Iterate over the graded components of `self`.

A graded component is a set of elements of the same depth.

The enumeration remembers only the last two graded components generated since the structure is symmetric.

OUTPUT:

An iterator of sets.

EXAMPLES:

```
sage: f = lambda a: [a-1, a+1]
sage: S = RecursivelyEnumeratedSet([10], f, structure='symmetric')
sage: it = S.graded_component_iterator()
sage: [sorted(next(it)) for _ in range(5)]
[[10], [9, 11], [8, 12], [7, 13], [6, 14]]
```

Starting with two generators:

```
sage: f = lambda a: [a-1, a+1]
sage: S = RecursivelyEnumeratedSet([5, 10], f, structure='symmetric')
sage: it = S.graded_component_iterator()
sage: [sorted(next(it)) for _ in range(5)]
[[5, 10], [4, 6, 9, 11], [3, 7, 8, 12], [2, 13], [1, 14]]
```

Gaussian integers:

```
sage: f = lambda a: [a+1, a+I]
sage: S = RecursivelyEnumeratedSet([0], f, structure='symmetric')
sage: it = S.graded_component_iterator()
sage: [sorted(next(it)) for _ in range(7)]
[[0],
 [I, 1],
 [2*I, I + 1, 2],
 [3*I, 2*I + 1, I + 2, 3],
 [4*I, 3*I + 1, 2*I + 2, I + 3, 4],
 [5*I, 4*I + 1, 3*I + 2, 2*I + 3, I + 4, 5],
 [6*I, 5*I + 1, 4*I + 2, 3*I + 3, 2*I + 4, I + 5, 6]]
```

CHAPTER

TWENTYEIGHT

MAPS BETWEEN FINITE SETS

This module implements parents modeling the set of all maps between two finite sets. At the user level, any such parent should be constructed using the factory class FiniteSetMaps which properly selects which of its subclasses to use.

AUTHORS:

- Florent Hivert

class sage.sets.finite_set_maps.**FiniteSetEndoMaps_N**(*n*, *action*, *category=None*)

Bases: sage.sets.finite_set_maps.FiniteSetMaps_MN

The sets of all maps from $\{1, 2, \ldots, n\}$ to itself

Users should use the factory class FiniteSetMaps to create instances of this class.

INPUT:

- n – an integer.
- category – the category in which the sets of maps is constructed. It must be a sub-category of FiniteMonoids() which is the default value.

Element

alias of FiniteSetEndoMap_N

an_element()

Returns a map in self

EXAMPLES:

```
sage: M = FiniteSetMaps(4)
sage: M.an_element()
[3, 2, 1, 0]
```

one()

EXAMPLES:

```
sage: M = FiniteSetMaps(4)
sage: M.one()
[0, 1, 2, 3]
```

class sage.sets.finite_set_maps.**FiniteSetEndoMaps_Set**(*domain*, *action*, *category=None*)

Bases: sage.sets.finite_set_maps.FiniteSetMaps_Set, sage.sets.finite_set_maps.FiniteSetEnc

The sets of all maps from a set to itself

Users should use the factory class FiniteSetMaps to create instances of this class.

INPUT:

- `domain` – an object in the category `FiniteSets()`.
- `category` – the category in which the sets of maps is constructed. It must be a sub-category of `FiniteMonoids()` which is the default value.

Element
alias of `FiniteSetEndoMap_Set`

class `sage.sets.finite_set_maps.`**`FiniteSetMaps`**

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.Parent`

Maps between finite sets

Constructs the set of all maps between two sets. The sets can be given using any of the three following ways:

1. an object in the category `Sets()`.
2. a finite iterable. In this case, an object of the class `FiniteEnumeratedSet` is constructed from the iterable.
3. an integer `n` designing the set $\{1, 2, \ldots, n\}$. In this case an object of the class `IntegerRange` is constructed.

INPUT:

- `domain` – a set, finite iterable, or integer.
- `codomain` – a set, finite iterable, integer, or `None` (default). In this last case, the maps are endo-maps of the domain.
- `action` – `"left"` (default) or `"right"`. The side where the maps act on the domain. This is used in particular to define the meaning of the product (composition) of two maps.
- `category` – the category in which the sets of maps is constructed. By default, this is `FiniteMonoids()` if the domain and codomain coincide, and `FiniteEnumeratedSets()` otherwise.

OUTPUT:

an instance of a subclass of `FiniteSetMaps` modeling the set of all maps between `domain` and `codomain`.

EXAMPLES:

We construct the set `M` of all maps from $\{a, b\}$ to $\{3, 4, 5\}$:

```
sage: M = FiniteSetMaps(["a", "b"], [3, 4, 5]); M
Maps from {'a', 'b'} to {3, 4, 5}
sage: M.cardinality()
9
sage: M.domain()
{'a', 'b'}
sage: M.codomain()
{3, 4, 5}
sage: for f in M: print f
map: a -> 3, b -> 3
map: a -> 3, b -> 4
map: a -> 3, b -> 5
map: a -> 4, b -> 3
map: a -> 4, b -> 4
map: a -> 4, b -> 5
map: a -> 5, b -> 3
map: a -> 5, b -> 4
map: a -> 5, b -> 5
```

Elements can be constructed from functions and dictionaries:

```
sage: M(lambda c: ord(c)-94)
map: a -> 3, b -> 4

sage: M.from_dict({'a':3, 'b':5})
map: a -> 3, b -> 5
```

If the domain is equal to the codomain, then maps can be composed:

```
sage: M = FiniteSetMaps([1, 2, 3])
sage: f = M.from_dict({1:2, 2:1, 3:3}); f
map: 1 -> 2, 2 -> 1, 3 -> 3
sage: g = M.from_dict({1:2, 2:3, 3:1}); g
map: 1 -> 2, 2 -> 3, 3 -> 1

sage: f * g
map: 1 -> 1, 2 -> 3, 3 -> 2
```

This makes M into a monoid:

```
sage: M.category()
Category of finite monoids
sage: M.one()
map: 1 -> 1, 2 -> 2, 3 -> 3
```

By default, composition is from right to left, which corresponds to an action on the left. If one specifies `action` to right, then the composition is from left to right:

```
sage: M = FiniteSetMaps([1, 2, 3], action = 'right')
sage: f = M.from_dict({1:2, 2:1, 3:3})
sage: g = M.from_dict({1:2, 2:3, 3:1})
sage: f * g
map: 1 -> 3, 2 -> 2, 3 -> 1
```

If the domains and codomains are both of the form $\{0, \dots\}$, then one can use the shortcut:

```
sage: M = FiniteSetMaps(2,3); M
Maps from {0, 1} to {0, 1, 2}
sage: M.cardinality()
9
```

For a compact notation, the elements are then printed as lists $[f(i), i = 0, \dots]$:

```
sage: list(M)
[[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]
```

TESTS:

```
sage: TestSuite(FiniteSetMaps(0)).run()
sage: TestSuite(FiniteSetMaps(0, 2)).run()
sage: TestSuite(FiniteSetMaps(2, 0)).run()
sage: TestSuite(FiniteSetMaps([], [])).run()
sage: TestSuite(FiniteSetMaps([1, 2], [])).run()
sage: TestSuite(FiniteSetMaps([], [1, 2])).run()
```

cardinality()

The cardinality of `self`

EXAMPLES:

```
sage: FiniteSetMaps(4, 3).cardinality()
81
```

class sage.sets.finite_set_maps.**FiniteSetMaps_MN**(*m*, *n*, *category=None*)

Bases: sage.sets.finite_set_maps.FiniteSetMaps

The set of all maps from $\{1, 2, \ldots, m\}$ to $\{1, 2, \ldots, n\}$.

Users should use the factory class FiniteSetMaps to create instances of this class.

INPUT:

- m, n – integers
- category – the category in which the sets of maps is constructed. It must be a sub-category of FiniteEnumeratedSets() which is the default value.

Element

alias of FiniteSetMap_MN

an_element()

Returns a map in self

EXAMPLES:

```
sage: M = FiniteSetMaps(4, 2)
sage: M.an_element()
[0, 0, 0, 0]

sage: M = FiniteSetMaps(0, 0)
sage: M.an_element()
[]
```

An exception EmptySetError is raised if this set is empty, that is if the codomain is empty and the domain is not.

sage: M = FiniteSetMaps(4, 0) sage: M.cardinality() 0 sage: M.an_element() Traceback (most recent call last): ... EmptySetError

codomain()

The codomain of self

EXAMPLES:

```
sage: FiniteSetMaps(3,2).codomain()
{0, 1}
```

domain()

The domain of self

EXAMPLES:

```
sage: FiniteSetMaps(3,2).domain()
{0, 1, 2}
```

class sage.sets.finite_set_maps.**FiniteSetMaps_Set**(*domain*, *codomain*, *category=None*)

Bases: sage.sets.finite_set_maps.FiniteSetMaps_MN

The sets of all maps between two sets

Users should use the factory class FiniteSetMaps to create instances of this class.

INPUT:

- `domain` – an object in the category `FiniteSets()`.
- `codomain` – an object in the category `FiniteSets()`.
- `category` – the category in which the sets of maps is constructed. It must be a sub-category of `FiniteEnumeratedSets()` which is the default value.

Element
alias of `FiniteSetMap_Set`

codomain()
The codomain of `self`

EXAMPLES:

```
sage: FiniteSetMaps(["a", "b"], [3, 4, 5]).codomain()
{3, 4, 5}
```

domain()
The domain of `self`

EXAMPLES:

```
sage: FiniteSetMaps(["a", "b"], [3, 4, 5]).domain()
{'a', 'b'}
```

from_dict(*d*)
Create a map from a dictionary

EXAMPLES:

```
sage: M = FiniteSetMaps(["a", "b"], [3, 4, 5])
sage: M.from_dict({"a": 4, "b": 3})
map: a -> 4, b -> 3
```

CHAPTER

TWENTYNINE

DATA STRUCTURES FOR MAPS BETWEEN FINITE SETS

This module implements several fast Cython data structures for maps between two finite set. Those classes are not intended to be used directly. Instead, such a map should be constructed via its parent, using the class `FiniteSetMaps`.

EXAMPLES:

To create a map between two sets, one first creates the set of such maps:

```
sage: M = FiniteSetMaps(["a", "b"], [3, 4, 5])
```

The map can then be constructed either from a function:

```
sage: f1 = M(lambda c: ord(c)-94); f1
map: a -> 3, b -> 4
```

or from a dictionary:

```
sage: f2 = M.from_dict({'a':3, 'b':4}); f2
map: a -> 3, b -> 4
```

The two created maps are equal:

```
sage: f1 == f2
True
```

Internally, maps are represented as the list of the ranks of the images `f(x)` in the co-domain, in the order of the domain:

```
sage: list(f2)
[0, 1]
```

A third fast way to create a map it to use such a list. it should be kept for internal use:

```
sage: f3 = M._from_list_([0, 1]); f3
map: a -> 3, b -> 4
sage: f1 == f3
True
```

AUTHORS:

- Florent Hivert

class `sage.sets.finite_set_map_cy.`**`FiniteSetEndoMap_N`**

Bases: `sage.sets.finite_set_map_cy.FiniteSetMap_MN`

Maps from `range(n)` to itself.

See also:

FiniteSetMap_MN for assumptions on the parent

TESTS:

```
sage: fs = FiniteSetMaps(3)([1, 0, 2])
sage: TestSuite(fs).run()
```

class sage.sets.finite_set_map_cy.**FiniteSetEndoMap_Set**

Bases: sage.sets.finite_set_map_cy.FiniteSetMap_Set

Maps from a set to itself

See also:

FiniteSetMap_Set for assumptions on the parent

TESTS:

```
sage: F = FiniteSetMaps(["a", "b", "c"])
sage: f = F.from_dict({"a": "b", "b": "a", "c": "b"}); f
map: a -> b, b -> a, c -> b
sage: TestSuite(f).run()
```

class sage.sets.finite_set_map_cy.**FiniteSetMap_MN**

Bases: sage.structure.list_clone.ClonableIntArray

Data structure for maps from `range(m)` to `range(n)`.

We assume that the parent given as argument is such that:

- `m` is stored in `self.parent()._m`
- `n` is stored in `self.parent()._n`
- the domain is in `self.parent().domain()`
- the codomain is in `self.parent().codomain()`

check()

Performs checks on `self`

Check that `self` is a proper function and then calls `parent.check_element(self)` where `parent` is the parent of `self`.

TESTS:

```
sage: fs = FiniteSetMaps(3, 2)
sage: for el in fs: el.check()
sage: fs([1,1])
Traceback (most recent call last):
...
AssertionError: Wrong number of values
sage: fs([0,0,2])
Traceback (most recent call last):
...
AssertionError: Wrong value self(2) = 2
```

codomain()

Returns the codomain of `self`

EXAMPLES:

```
sage: FiniteSetMaps(4, 3)([1, 0, 2, 1]).codomain()
{0, 1, 2}
```

domain()

Returns the domain of `self`

EXAMPLES:

```
sage: FiniteSetMaps(4, 3)([1, 0, 2, 1]).domain()
{0, 1, 2, 3}
```

fibers()

Returns the fibers of `self`

OUTPUT:

a dictionary `d` such that `d[y]` is the set of all `x` in `domain` such that `f(x) = y`

EXAMPLES:

```
sage: FiniteSetMaps(4, 3)([1, 0, 2, 1]).fibers()
{0: {1}, 1: {0, 3}, 2: {2}}
sage: F = FiniteSetMaps(["a", "b", "c"])
sage: F.from_dict({"a": "b", "b": "a", "c": "b"}).fibers()
{'a': {'b'}, 'b': {'a', 'c'}}
```

getimage(*i*)

Returns the image of `i` by `self`

INPUT:

- `i` – any object.

Note: if you need speed, please use instead `_getimage()`

EXAMPLES:

```
sage: fs = FiniteSetMaps(4, 3)([1, 0, 2, 1])
sage: fs.getimage(0), fs.getimage(1), fs.getimage(2), fs.getimage(3)
(1, 0, 2, 1)
```

image_set()

Returns the image set of `self`

EXAMPLES:

```
sage: FiniteSetMaps(4, 3)([1, 0, 2, 1]).image_set()
{0, 1, 2}
sage: FiniteSetMaps(4, 3)([1, 0, 0, 1]).image_set()
{0, 1}
```

items()

The items of `self`

Return the list of the ordered pairs `(x, self(x))`

EXAMPLES:

```
sage: FiniteSetMaps(4, 3)([1, 0, 2, 1]).items()
[(0, 1), (1, 0), (2, 2), (3, 1)]
```

setimage(*i, j*)

Set the image of `i` as `j` in `self`

Warning: `self` must be mutable; otherwise an exception is raised.

INPUT:

- i, j – two object's

OUTPUT: None

Note: if you need speed, please use instead _setimage()

EXAMPLES:

```
sage: fs = FiniteSetMaps(4, 3)([1, 0, 2, 1])
sage: fs2 = copy(fs)
sage: fs2.setimage(2, 1)
sage: fs2
[1, 0, 1, 1]
sage: with fs.clone() as fs3:
...       fs3.setimage(0, 2)
...       fs3.setimage(1, 2)
sage: fs3
[2, 2, 2, 1]
```

class sage.sets.finite_set_map_cy.**FiniteSetMap_Set**

Bases: sage.sets.finite_set_map_cy.FiniteSetMap_MN

Data structure for maps

We assume that the parent given as argument is such that:

- the domain is in parent.domain()
- the codomain is in parent.codomain()
- parent._m contains the cardinality of the domain
- parent._n contains the cardinality of the codomain
- parent._unrank_domain and parent._rank_domain is a pair of reciprocal rank and unrank functions beween the domain and range(parent._m).
- parent._unrank_codomain and parent._rank_codomain is a pair of reciprocal rank and unrank functions beween the codomain and range(parent._n).

classmethod from_dict (*parent*, *d*)

Creates a FiniteSetMap from a dictionary

Warning: no check is performed !

TESTS:

```
sage: from sage.sets.finite_set_map_cy import FiniteSetMap_Set_from_dict as from_dict
sage: F = FiniteSetMaps(["a", "b"], [3, 4, 5])
sage: f = from_dict(F.element_class, F, {"a": 3, "b": 5}); f.check(); f
map: a -> 3, b -> 5
sage: f.parent() is F
True
sage: f.is_immutable()
True
```

classmethod from_list (*parent*, *lst*)

Creates a FiniteSetMap from a list

Warning: no check is performed !

TESTS:

```
sage: from sage.sets.finite_set_map_cy import FiniteSetMap_Set_from_list as from_list
sage: F = FiniteSetMaps(["a", "b"], [3, 4, 5])
sage: f = from_list(F.element_class, F, [0,2]); f.check(); f
map: a -> 3, b -> 5
sage: f.parent() is F
True
sage: f.is_immutable()
True
```

getimage (*i*)

Returns the image of `i` by `self`

INPUT:

- `i` – an `int`

EXAMPLES:

```
sage: F = FiniteSetMaps(["a", "b", "c", "d"], ["u", "v", "w"])
sage: fs = F._from_list_([1, 0, 2, 1])
sage: map(fs.getimage, ["a", "b", "c", "d"])
['v', 'u', 'w', 'v']
```

image_set ()

Returns the image set of `self`

EXAMPLES:

```
sage: F = FiniteSetMaps(["a", "b", "c"])
sage: F.from_dict({"a": "b", "b": "a", "c": "b"}).image_set()
{'a', 'b'}
sage: F = FiniteSetMaps(["a", "b", "c"])
sage: F(lambda x: "c").image_set()
{'c'}
```

items ()

The items of `self`

Return the list of the couple `(x, self(x))`

EXAMPLES:

```
sage: F = FiniteSetMaps(["a", "b", "c"])
sage: F.from_dict({"a": "b", "b": "a", "c": "b"}).items()
[('a', 'b'), ('b', 'a'), ('c', 'b')]
```

TESTS:

```
sage: all(F.from_dict(dict(f.items())) == f for f in F)
True
```

setimage (*i, j*)

Set the image of `i` as `j` in `self`

Warning: `self` must be mutable otherwise an exception is raised.

INPUT:

•i, j – two object‘s

OUTPUT: None

EXAMPLES:

```
sage: F = FiniteSetMaps(["a", "b", "c", "d"], ["u", "v", "w"])
sage: fs = F(lambda x: "v")
sage: fs2 = copy(fs)
sage: fs2.setimage("a", "w")
sage: fs2
map: a -> w, b -> v, c -> v, d -> v
sage: with fs.clone() as fs3:
...       fs3.setimage("a", "u")
...       fs3.setimage("c", "w")
sage: fs3
map: a -> u, b -> v, c -> w, d -> v
```

TESTS:

```
sage: with fs.clone() as fs3:
...       fs3.setimage("z", 2)
Traceback (most recent call last):
...
ValueError: 'z' is not in list

sage: with fs.clone() as fs3:
...       fs3.setimage(1, 4)
Traceback (most recent call last):
...
ValueError: 1 is not in list
```

sage.sets.finite_set_map_cy.**FiniteSetMap_Set_from_dict**(*cls*, *parent*, *d*)

Creates a FiniteSetMap from a dictionary

Warning: no check is performed !

TESTS:

```
sage: from sage.sets.finite_set_map_cy import FiniteSetMap_Set_from_dict as from_dict
sage: F = FiniteSetMaps(["a", "b"], [3, 4, 5])
sage: f = from_dict(F.element_class, F, {"a": 3, "b": 5}); f.check(); f
map: a -> 3, b -> 5
sage: f.parent() is F
True
sage: f.is_immutable()
True
```

sage.sets.finite_set_map_cy.**FiniteSetMap_Set_from_list**(*cls*, *parent*, *lst*)

Creates a FiniteSetMap from a list

Warning: no check is performed !

TESTS:

```
sage: from sage.sets.finite_set_map_cy import FiniteSetMap_Set_from_list as from_list
sage: F = FiniteSetMaps(["a", "b"], [3, 4, 5])
sage: f = from_list(F.element_class, F, [0,2]); f.check(); f
map: a -> 3, b -> 5
```

```
sage: f.parent() is F
True
sage: f.is_immutable()
True
```

sage.sets.finite_set_map_cy.**fibers**(*f*, *domain*)
Returns the fibers of the function `f` on the finite set `domain`

INPUT:

- `f` – a function or callable
- `domain` – a finite iterable

OUTPUT:

- a dictionary `d` such that `d[y]` is the set of all `x` in `domain` such that `f(x) = y`

EXAMPLES:

```
sage: from sage.sets.finite_set_map_cy import fibers, fibers_args
sage: fibers(lambda x: 1, [])
{}
sage: fibers(lambda x: x^2, [-1, 2, -3, 1, 3, 4])
{1: {1, -1}, 4: {2}, 9: {3, -3}, 16: {4}}
sage: fibers(lambda x: 1,   [-1, 2, -3, 1, 3, 4])
{1: {1, 2, 3, 4, -3, -1}}
sage: fibers(lambda x: 1, [1,1,1])
{1: {1}}
```

See also:

`fibers_args()` if one needs to pass extra arguments to `f`.

sage.sets.finite_set_map_cy.**fibers_args**(*f*, *domain*, **args*, ***opts*)
Returns the fibers of the function `f` on the finite set `domain`

It is the same as `fibers()` except that one can pass extra argument for `f` (with a small overhead)

EXAMPLES:

```
sage: from sage.sets.finite_set_map_cy import fibers_args
sage: fibers_args(operator.pow, [-1, 2, -3, 1, 3, 4], 2)
{1: {1, -1}, 4: {2}, 9: {3, -3}, 16: {4}}
```

CHAPTER

THIRTY

INTEGER RANGE

AUTHORS:

- Nicolas Borie (2010-03): First release.
- Florent Hivert (2010-03): Added a class factory + cardinality method.
- Vincent Delecroix (2012-02): add methods rank/unrank, make it complient with Python int.

class `sage.sets.integer_range.`**`IntegerRange`**

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.Parent`

The class of `Integer` ranges

Returns an enumerated set containing an arithmetic progression of integers.

INPUT:

- `begin` – an integer, Infinity or -Infinity
- `end` – an integer, Infinity or -Infinity
- `step` – a non zero integer (default to 1)
- `middle_point` – an integer inside the set (default to `None`)

OUTPUT:

A parent in the category `FiniteEnumeratedSets()` or `InfiniteEnumeratedSets()` depending on the arguments defining `self`.

`IntegerRange(i, j)` returns the set of $\{i, i+1, i+2, \dots, j-1\}$. `start` (!) defaults to 0. When `step` is given, it specifies the increment. The default increment is 1. IntegerRange allows `begin` and `end` to be infinite.

`IntegerRange` is designed to have similar interface Python range. However, whereas `range` accept and returns Python `int`, `IntegerRange` deals with `Integer`.

If `middle_point` is given, then the elements are generated starting from it, in a alternating way: $\{m, m+1, m-2, m+2, m-2\dots\}$.

EXAMPLES:

```
sage: list(IntegerRange(5))
[0, 1, 2, 3, 4]
sage: list(IntegerRange(2,5))
[2, 3, 4]
sage: I = IntegerRange(2,100,5); I
{2, 7, ..., 97}
sage: list(I)
[2, 7, 12, 17, 22, 27, 32, 37, 42, 47, 52, 57, 62, 67, 72, 77, 82, 87, 92, 97]
```

```
sage: I.category()
Category of facade finite enumerated sets
sage: I[1].parent()
Integer Ring
```

When `begin` and `end` are both finite, `IntegerRange(begin, end, step)` is the set whose list of elements is equivalent to the python construction `range(begin, end, step)`:

```
sage: list(IntegerRange(4,105,3)) == range(4,105,3)
True
sage: list(IntegerRange(-54,13,12)) == range(-54,13,12)
True
```

Except for the type of the numbers:

```
sage: type(IntegerRange(-54,13,12)[0]), type(range(-54,13,12)[0])
(<type 'sage.rings.integer.Integer'>, <type 'int'>)
```

When `begin` is finite and `end` is +Infinity, `self` is the infinite arithmetic progression starting from the `begin` by step `step`:

```
sage: I = IntegerRange(54,Infinity,3); I
{54, 57, ...}
sage: I.category()
Category of facade infinite enumerated sets
sage: p = iter(I)
sage: (next(p), next(p), next(p), next(p), next(p), next(p))
(54, 57, 60, 63, 66, 69)

sage: I = IntegerRange(54,-Infinity,-3); I
{54, 51, ...}
sage: I.category()
Category of facade infinite enumerated sets
sage: p = iter(I)
sage: (next(p), next(p), next(p), next(p), next(p), next(p))
(54, 51, 48, 45, 42, 39)
```

When `begin` and `end` are both infinite, you will have to specify the extra argument `middle_point`. `self` is then defined by a point and a progression/regression setting by `step`. The enumeration is done this way: (let us call m the `middle_point`) $\{m, m + step, m - step, m + 2step, m - 2step, m + 3step, \dots\}$:

```
sage: I = IntegerRange(-Infinity,Infinity,37,-12); I
Integer progression containing -12 with increment 37 and bounded with -Infinity and +Infinity
sage: I.category()
Category of facade infinite enumerated sets
sage: -12 in I
True
sage: -15 in I
False
sage: p = iter(I)
sage: (next(p), next(p), next(p), next(p), next(p), next(p), next(p), next(p))
(-12, 25, -49, 62, -86, 99, -123, 136)
```

It is also possible to use the argument `middle_point` for other cases, finite or infinite. The set will be the same as if you didn't give this extra argument but the enumeration will begin with this `middle_point`:

```
sage: I = IntegerRange(123,-12,-14); I
{123, 109, ..., -3}
sage: list(I)
[123, 109, 95, 81, 67, 53, 39, 25, 11, -3]
```

```
sage: J = IntegerRange(123,-12,-14,25); J
Integer progression containing 25 with increment -14 and bounded with 123 and -12
sage: list(J)
[25, 11, 39, -3, 53, 67, 81, 95, 109, 123]
```

Remember that, like for range, if you define a non empty set, `begin` is supposed to be included and `end` is supposed to be excluded. In the same way, when you define a set with a `middle_point`, the `begin` bound will be supposed to be included and the `end` bound supposed to be excluded:

```
sage: I = IntegerRange(-100,100,10,0)
sage: J = range(-100,100,10)
sage: 100 in I
False
sage: 100 in J
False
sage: -100 in I
True
sage: -100 in J
True
sage: list(I)
[0, 10, -10, 20, -20, 30, -30, 40, -40, 50, -50, 60, -60, 70, -70, 80, -80, 90, -90, -100]
```

Note: The input is normalized so that:

```
sage: IntegerRange(1, 6, 2) is IntegerRange(1, 7, 2)
True
sage: IntegerRange(1, 8, 3) is IntegerRange(1, 10, 3)
True
```

TESTS:

```
sage: # Some category automatic tests
sage: TestSuite(IntegerRange(2,100,3)).run()
sage: TestSuite(IntegerRange(564,-12,-46)).run()
sage: TestSuite(IntegerRange(2,Infinity,3)).run()
sage: TestSuite(IntegerRange(732,-Infinity,-13)).run()
sage: TestSuite(IntegerRange(-Infinity,Infinity,3,2)).run()
sage: TestSuite(IntegerRange(56,Infinity,12,80)).run()
sage: TestSuite(IntegerRange(732,-12,-2743,732)).run()
sage: # 20 random tests: range and IntegerRange give the same set for finite cases
sage: for i in range(20):
...       begin = Integer(randint(-300,300))
...       end = Integer(randint(-300,300))
...       step = Integer(randint(-20,20))
...       if step == 0:
...           step = Integer(1)
...       assert list(IntegerRange(begin, end, step)) == range(begin, end, step)
sage: # 20 random tests: range and IntegerRange with middle point for finite cases
sage: for i in range(20):
...       begin = Integer(randint(-300,300))
...       end = Integer(randint(-300,300))
...       step = Integer(randint(-15,15))
...       if step == 0:
...           step = Integer(-3)
...       I = IntegerRange(begin, end, step)
...       if I.cardinality() == 0:
...           assert len(range(begin, end, step)) == 0
...       else:
```

```
...             TestSuite(I).run()
...             L1 = list(IntegerRange(begin, end, step, I.an_element()))
...             L2 = range(begin, end, step)
...             L1.sort()
...             L2.sort()
...             assert L1 == L2
```

Thanks to trac ticket #8543 empty integer range are allowed:

```
sage: TestSuite(IntegerRange(0, 5, -1)).run()
```

element_class
alias of `Integer`

class `sage.sets.integer_range.`**`IntegerRangeEmpty`**(*elements*)
Bases: `sage.sets.integer_range.IntegerRange`, `sage.sets.finite_enumerated_set.FiniteEnumera`

A singleton class for empty integer ranges

See `IntegerRange` for more details.

class `sage.sets.integer_range.`**`IntegerRangeFinite`**(*begin*, *end*, *step=1*)
Bases: `sage.sets.integer_range.IntegerRange`

The class of finite enumerated sets of integers defined by finite arithmetic progressions

See `IntegerRange` for more details.

cardinality()
Return the cardinality of `self`

EXAMPLES:

```
sage: IntegerRange(123,12,-4).cardinality()
28
sage: IntegerRange(-57,12,8).cardinality()
9
sage: IntegerRange(123,12,4).cardinality()
0
```

rank(*x*)
EXAMPLES:

```
sage: I = IntegerRange(-57,36,8)
sage: I.rank(23)
10
sage: I.unrank(10)
23
sage: I.rank(22)
Traceback (most recent call last):
...
IndexError: 22 not in self
sage: I.rank(87)
Traceback (most recent call last):
...
IndexError: 87 not in self
```

unrank(*i*)
Return the i-th elt of this integer range.

EXAMPLES:

```
sage: I=IntegerRange(1,13,5)
sage: I[0], I[1], I[2]
(1, 6, 11)
sage: I[3]
Traceback (most recent call last):
...
IndexError: out of range
sage: I[-1]
11
sage: I[-4]
Traceback (most recent call last):
...
IndexError: out of range

sage: I = IntegerRange(13,1,-1)
sage: l = I.list()
sage: [I[i] for i in xrange(I.cardinality())] == l
True
sage: l.reverse()
sage: [I[i] for i in xrange(-1,-I.cardinality()-1,-1)] == l
True
```

class sage.sets.integer_range.**IntegerRangeFromMiddle**(*begin*, *end*, *step=1*, *middle_point=1*)

Bases: sage.sets.integer_range.IntegerRange

The class of finite or infinite enumerated sets defined with an inside point, a progression and two limits.

See IntegerRange for more details.

next(*elt*)

Return the next element of elt in self.

EXAMPLES:

```
sage: from sage.sets.integer_range import IntegerRangeFromMiddle
sage: I = IntegerRangeFromMiddle(-100,100,10,0)
sage: (I.next(0), I.next(10), I.next(-10), I.next(20), I.next(-100))
(10, -10, 20, -20, None)
sage: I = IntegerRangeFromMiddle(-Infinity,Infinity,10,0)
sage: (I.next(0), I.next(10), I.next(-10), I.next(20), I.next(-100))
(10, -10, 20, -20, 110)
sage: I.next(1)
Traceback (most recent call last):
...
LookupError: 1 not in Integer progression containing 0 with increment 10 and bounded with -I
```

class sage.sets.integer_range.**IntegerRangeInfinite**(*begin*, *step=1*)

Bases: sage.sets.integer_range.IntegerRange

The class of infinite enumerated sets of integers defined by infinite arithmetic progressions.

See IntegerRange for more details.

rank(*x*)

EXAMPLES:

```
sage: I = IntegerRange(-57,Infinity,8)
sage: I.rank(23)
10
sage: I.unrank(10)
```

```
23
sage: I.rank(22)
Traceback (most recent call last):
...
IndexError: 22 not in self
```

unrank (*i*)

Returns the i-th element of self.

EXAMPLES:

```
sage: I = IntegerRange(-8,Infinity,3)
sage: I.unrank(1)
-5
```

CHAPTER

THIRTYONE

POSITIVE INTEGERS

class sage.sets.positive_integers.**PositiveIntegers**

Bases: sage.sets.integer_range.IntegerRangeInfinite

The enumerated set of positive integers. To fix the ideas, we mean $\{1, 2, 3, 4, 5, \dots\}$.

This class implements the set of positive integers, as an enumerated set (see InfiniteEnumeratedSets).

This set is an integer range set. The construction is therefore done by IntegerRange (see IntegerRange).

EXAMPLES:

```
sage: PP = PositiveIntegers()
sage: PP
Positive integers
sage: PP.cardinality()
+Infinity
sage: TestSuite(PP).run()
sage: PP.list()
Traceback (most recent call last):
...
NotImplementedError: infinite list
sage: it = iter(PP)
sage: (next(it), next(it), next(it), next(it), next(it))
(1, 2, 3, 4, 5)
sage: PP.first()
1
```

TESTS:

```
sage: TestSuite(PositiveIntegers()).run()
```

an_element()

Returns an element of self.

EXAMPLES:

```
sage: PositiveIntegers().an_element()
42
```

CHAPTER

THIRTYTWO

NON NEGATIVE INTEGERS

class sage.sets.non_negative_integers.**NonNegativeIntegers**(*category=None*)

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

The enumerated set of non negative integers.

This class implements the set of non negative integers, as an enumerated set (see InfiniteEnumeratedSets).

EXAMPLES:

```
sage: NN = NonNegativeIntegers()
sage: NN
Non negative integers
sage: NN.cardinality()
+Infinity
sage: TestSuite(NN).run()
sage: NN.list()
Traceback (most recent call last):
...
NotImplementedError: infinite list
sage: NN.element_class
<type 'sage.rings.integer.Integer'>
sage: it = iter(NN)
sage: [next(it), next(it), next(it), next(it), next(it)]
[0, 1, 2, 3, 4]
sage: NN.first()
0
```

Currently, this is just a "facade" parent; namely its elements are plain Sage Integers with Integer Ring as parent:

```
sage: x = NN(15); type(x)
<type 'sage.rings.integer.Integer'>
sage: x.parent()
Integer Ring
sage: x+3
18
```

In a later version, there will be an option to specify whether the elements should have Integer Ring or Non negative integers as parent:

```
sage: NN = NonNegativeIntegers(facade = False) # todo: not implemented
sage: x = NN(5)                                # todo: not implemented
sage: x.parent()                               # todo: not implemented
Non negative integers
```

This runs generic sanity checks on NN:

```
sage: TestSuite(NN).run()
```

TODO: do not use NN any more in the doctests for NonNegativeIntegers.

Element
alias of Integer

an_element()
EXAMPLES:

```
sage: NonNegativeIntegers().an_element()
42
```

from_integer
alias of Integer

next(*o*)
EXAMPLES:

```
sage: NN = NonNegativeIntegers()
sage: NN.next(3)
4
```

some_elements()
EXAMPLES:

```
sage: NonNegativeIntegers().some_elements()
[0, 1, 3, 42]
```

unrank(*rnk*)
EXAMPLES:

```
sage: NN = NonNegativeIntegers()
sage: NN.unrank(100)
100
```

CHAPTER

THIRTYTHREE

THE SET OF PRIME NUMBERS

AUTHORS:

- William Stein (2005): original version
- Florent Hivert (2009-11): adapted to the category framework. The following methods were removed:
  - cardinality, __len__, __iter__: provided by EnumeratedSets
  - __cmp__(self, other): __eq__ is provided by UniqueRepresentation and seems to do as good a job (all test pass)

class sage.sets.primes.**Primes**(*proof*)

Bases: sage.structure.parent.Set_generic, sage.structure.unique_representation.UniqueRepre

The set of prime numbers.

EXAMPLES:

```
sage: P = Primes(); P
Set of all prime numbers: 2, 3, 5, 7, ...
```

We show various operations on the set of prime numbers:

```
sage: P.cardinality()
+Infinity
sage: R = Primes()
sage: P == R
True
sage: 5 in P
True
sage: 100 in P
False

sage: len(P)          # note: this used to be a TypeError
Traceback (most recent call last):
...
NotImplementedError: infinite list
```

first()

Returns the first prime number.

EXAMPLES:

```
sage: P = Primes()
sage: P.first()
2
```

next (*pr*)
Returns the next prime number.

EXAMPLES:

```
sage: P = Primes()
sage: P.next(5)
7
```

unrank (*n*)
Returns the n-th prime number.

EXAMPLES:: sage: P = Primes() sage: P.unrank(0) 2 sage: P.unrank(5) 13 sage: P.unrank(42) 191

CHAPTER

THIRTYFOUR

TOTALLY ORDERED FINITE SETS

AUTHORS:

- Stepan Starosta (2012): Initial version

class `sage.sets.totally_ordered_finite_set.`**`TotallyOrderedFiniteSet`**(*elements*, *facade=True*)

Bases: `sage.sets.finite_enumerated_set.FiniteEnumeratedSet`

Totally ordered finite set.

This is a finite enumerated set assuming that the elements are ordered based upon their rank (i.e. their position in the set).

INPUT:

- `elements` – A list of elements in the set
- `facade` – (default: `True`) if `True`, a facade is used; it should be set to `False` if the elements do not inherit from `Element` or if you want a funny order. See examples for more details.

See also:

`FiniteEnumeratedSet`

EXAMPLES:

```
sage: S = TotallyOrderedFiniteSet([1,2,3])
sage: S
{1, 2, 3}
sage: S.cardinality()
3
```

By default, totally ordered finite set behaves as a facade:

```
sage: S(1).parent()
Integer Ring
```

It makes comparison fails when it is not the standard order:

```
sage: T1 = TotallyOrderedFiniteSet([3,2,5,1])
sage: T1(3) < T1(1)
False
sage: T2 = TotallyOrderedFiniteSet([3,var('x')])
sage: T2(3) < T2(var('x'))
3 < x
```

To make the above example work, you should set the argument facade to `False` in the constructor. In that case, the elements of the set have a dedicated class:

```
sage: A = TotallyOrderedFiniteSet([3,2,0,'a',7,(0,0),1], facade=False)
sage: A
{3, 2, 0, 'a', 7, (0, 0), 1}
sage: x = A.an_element()
sage: x
3
sage: x.parent()
{3, 2, 0, 'a', 7, (0, 0), 1}
sage: A(3) < A(2)
True
sage: A('a') < A(7)
True
sage: A(3) > A(2)
False
sage: A(1) < A(3)
False
sage: A(3) == A(3)
True
```

But then, the equality comparison is always False with elements outside of the set:

```
sage: A(1) == 1
False
sage: 1 == A(1)
False
sage: 'a' == A('a')
False
sage: A('a') == 'a'
False
```

and comparisons are comparisons of types:

```
sage: for e in [1,'a',(0, 0)]:
...       f = A(e)
...       print e == f,
...       print cmp(e,f) == cmp(type(e),type(f)),
...       print cmp(f,e) == cmp(type(f),type(e))
False True True
False True True
False True True
```

This behavior of comparison is the same as the one of `Element`.

Since trac ticket #16280, totally ordered sets support elements that do not inherit from `sage.structure.element.Element`, whether they are facade or not:

```
sage: S = TotallyOrderedFiniteSet(['a','b'])
sage: S('a')
'a'
sage: S = TotallyOrderedFiniteSet(['a','b'], facade = False)
sage: S('a')
'a'
```

Multiple elements are automatically deleted:

```
sage: TotallyOrderedFiniteSet([1,1,2,1,2,2,5,4])
{1, 2, 5, 4}
```

Element

alias of `TotallyOrderedFiniteSetElement`

le (*x*, *y*)

Return `True` if $x \leq y$ for the order of `self`.

EXAMPLES:

```
sage: T = TotallyOrderedFiniteSet([1,3,2], facade=False)
sage: T1, T3, T2 = T.list()
sage: T.le(T1,T3)
True
sage: T.le(T3,T2)
True
```

class `sage.sets.totally_ordered_finite_set.`**`TotallyOrderedFiniteSetElement`** (*parent*, *data*)

Bases: `sage.structure.element.Element`

Element of a finite totally ordered set.

EXAMPLES:

```
sage: S = TotallyOrderedFiniteSet([2,7], facade=False)
sage: x = S(2)
sage: print x
2
sage: x.parent()
{2, 7}
```

CHAPTER

THIRTYFIVE

SUBSETS OF THE REAL LINE

This module contains subsets of the real line that can be constructed as the union of a finite set of open and closed intervals.

EXAMPLES:

```
sage: RealSet(0,1)
(0, 1)
sage: RealSet((0,1), [2,3])
(0, 1) + [2, 3]
sage: RealSet(-oo, oo)
(-oo, +oo)
```

Brackets must be balanced in Python, so the naive notation for half-open intervals does not work:

```
sage: RealSet([0,1))
Traceback (most recent call last):
...
SyntaxError: invalid syntax
```

Instead, you can use the following construction functions:

```
sage: RealSet.open_closed(0,1)
(0, 1]
sage: RealSet.closed_open(0,1)
[0, 1)
sage: RealSet.point(1/2)
{1/2}
sage: RealSet.unbounded_below_open(0)
(-oo, 0)
sage: RealSet.unbounded_below_closed(0)
(-oo, 0]
sage: RealSet.unbounded_above_open(1)
(1, +oo)
sage: RealSet.unbounded_above_closed(1)
[1, +oo)
```

AUTHORS:

- Laurent Claessens (2010-12-10): Interval and ContinuousSet, posted to sage-devel at http://www.mail-archive.com/sage-support@googlegroups.com/msg21326.html.
- Ares Ribo (2011-10-24): Extended the previous work defining the class RealSet.
- Jordi Saludes (2011-12-10): Documentation and file reorganization.
- Volker Braun (2013-06-22): Rewrite

class sage.sets.real_set.**InternalRealInterval**(*lower*, *lower_closed*, *upper*, *upper_closed*, *check=True*)

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

A real interval.

You are not supposed to create RealInterval objects yourself. Always use RealSet instead.

INPUT:

- lower – real or minus infinity; the lower bound of the interval.
- lower_closed – boolean; whether the interval is closed at the lower bound
- upper – real or (plus) infinity; the upper bound of the interval
- upper_closed – boolean; whether the interval is closed at the upper bound
- check – boolean; whether to check the other arguments for validity

closure()

Return the closure

OUTPUT:

The closure as a new RealInterval

EXAMPLES:

```
sage: RealSet.open(0,1)[0].closure()
[0, 1]
sage: RealSet.open(-oo,1)[0].closure()
(-oo, 1]
sage: RealSet.open(0, oo)[0].closure()
[0, +oo)
```

contains(*x*)

Return whether x is contained in the interval

INPUT:

- x – a real number.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: i = RealSet.open_closed(0,2)[0]; i
(0, 2]
sage: i.contains(0)
False
sage: i.contains(1)
True
sage: i.contains(2)
True
```

convex_hull(*other*)

Return the convex hull of the two intervals

OUTPUT:

The convex hull as a new RealInterval.

EXAMPLES:

```
sage: I1 = RealSet.open(0, 1)[0];  I1
(0, 1)
sage: I2 = RealSet.closed(1, 2)[0];  I2
[1, 2]
sage: I1.convex_hull(I2)
(0, 2]
sage: I2.convex_hull(I1)
(0, 2]
sage: I1.convex_hull(I2.interior())
(0, 2)
sage: I1.closure().convex_hull(I2.interior())
[0, 2)
sage: I1.closure().convex_hull(I2)
[0, 2]
sage: I3 = RealSet.closed(1/2, 3/2)[0]; I3
[1/2, 3/2]
sage: I1.convex_hull(I3)
(0, 3/2]
```

element_class

alias of `LazyFieldElement`

interior()

Return the interior

OUTPUT:

The interior as a new `RealInterval`

EXAMPLES:

```
sage: RealSet.closed(0, 1)[0].interior()
(0, 1)
sage: RealSet.open_closed(-oo, 1)[0].interior()
(-oo, 1)
sage: RealSet.closed_open(0, oo)[0].interior()
(0, +oo)
```

intersection(*other*)

Return the intersection of the two intervals

INPUT:

- `other` – a `RealInterval`

OUTPUT:

The intersection as a new `RealInterval`

EXAMPLES:

```
sage: I1 = RealSet.open(0, 2)[0];  I1
(0, 2)
sage: I2 = RealSet.closed(1, 3)[0];  I2
[1, 3]
sage: I1.intersection(I2)
[1, 2)
sage: I2.intersection(I1)
[1, 2)
sage: I1.closure().intersection(I2.interior())
(1, 2]
sage: I2.interior().intersection(I1.closure())
```

```
(1, 2]

sage: I3 = RealSet.closed(10, 11)[0];  I3
[10, 11]
sage: I1.intersection(I3)
(0, 0)
sage: I3.intersection(I1)
(0, 0)
```

is_connected(*other*)

Test whether two intervals are connected

OUTPUT:

Boolean. Whether the set-theoretic union of the two intervals has a single connected component.

EXAMPLES:

```
sage: I1 = RealSet.open(0, 1)[0];  I1
(0, 1)
sage: I2 = RealSet.closed(1, 2)[0];  I2
[1, 2]
sage: I1.is_connected(I2)
True
sage: I1.is_connected(I2.interior())
False
sage: I1.closure().is_connected(I2.interior())
True
sage: I2.is_connected(I1)
True
sage: I2.interior().is_connected(I1)
False
sage: I2.closure().is_connected(I1.interior())
True
sage: I3 = RealSet.closed(1/2, 3/2)[0]; I3
[1/2, 3/2]
sage: I1.is_connected(I3)
True
sage: I3.is_connected(I1)
True
```

is_empty()

Return whether the interval is empty

The normalized form of `RealSet` has all intervals non-empty, so this method usually returns `False`.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: I = RealSet(0, 1)[0]
sage: I.is_empty()
False
```

is_point()

Return whether the interval consists of a single point

OUTPUT:

Boolean.

EXAMPLES:

```
sage: I = RealSet(0, 1)[0]
sage: I.is_point()
False
```

lower()

Return the lower bound

OUTPUT:

The lower bound as it was originally specified.

EXAMPLES:

```
sage: I = RealSet(0, 1)[0]
sage: I.lower()
0
sage: I.upper()
1
```

lower_closed()

Return whether the interval is open at the lower bound

OUTPUT:

Boolean.

EXAMPLES:

```
sage: I = RealSet.open_closed(0, 1)[0];  I
(0, 1]
sage: I.lower_closed()
False
sage: I.lower_open()
True
sage: I.upper_closed()
True
sage: I.upper_open()
False
```

lower_open()

Return whether the interval is closed at the upper bound

OUTPUT:

Boolean.

EXAMPLES:

```
sage: I = RealSet.open_closed(0, 1)[0];  I
(0, 1]
sage: I.lower_closed()
False
sage: I.lower_open()
True
sage: I.upper_closed()
True
sage: I.upper_open()
False
```

upper()

Return the upper bound

OUTPUT:

The upper bound as it was originally specified.

EXAMPLES:

```
sage: I = RealSet(0, 1)[0]
sage: I.lower()
0
sage: I.upper()
1
```

upper_closed()

Return whether the interval is closed at the lower bound

OUTPUT:

Boolean.

EXAMPLES:

```
sage: I = RealSet.open_closed(0, 1)[0];  I
(0, 1]
sage: I.lower_closed()
False
sage: I.lower_open()
True
sage: I.upper_closed()
True
sage: I.upper_open()
False
```

upper_open()

Return whether the interval is closed at the upper bound

OUTPUT:

Boolean.

EXAMPLES:

```
sage: I = RealSet.open_closed(0, 1)[0];  I
(0, 1]
sage: I.lower_closed()
False
sage: I.lower_open()
True
sage: I.upper_closed()
True
sage: I.upper_open()
False
```

class sage.sets.real_set.**RealSet**(*intervals*)

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

A subset of the real line

INPUT:

Arguments defining a real set. Possibilities are either two real numbers to construct an open set or a list/tuple/iterable of intervals. The individual intervals can be specified by either a RealInterval, a tuple of two real numbers (constructing an open interval), or a list of two number (constructing a closed interval).

EXAMPLES:

```
sage: RealSet(0,1)     # open set from two numbers
(0, 1)
sage: i = RealSet(0,1)[0]
sage: RealSet(i)       # interval
(0, 1)
sage: RealSet(i, (3,4))     # tuple of two numbers = open set
(0, 1) + (3, 4)
sage: RealSet(i, [3,4])     # list of two numbers = closed set
(0, 1) + [3, 4]
```

an_element()

Return a point of the set

OUTPUT:

A real number. `ValueError` if the set is empty.

EXAMPLES:

```
sage: RealSet.open_closed(0, 1).an_element()
1
sage: RealSet(0, 1).an_element()
1/2
```

static **are_pairwise_disjoint**(*real_set_collection*)

Test whether sets are pairwise disjoint

INPUT:

- `*real_set_collection` – a list/tuple/iterable of `RealSet`.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: s1 = RealSet((0, 1), (2, 3))
sage: s2 = RealSet((1, 2))
sage: s3 = RealSet.point(3)
sage: RealSet.are_pairwise_disjoint(s1, s2, s3)
True
sage: RealSet.are_pairwise_disjoint(s1, s2, s3, [10,10])
True
sage: RealSet.are_pairwise_disjoint(s1, s2, s3, [-1, 1/2])
False
```

cardinality()

Return the cardinality of the subset of the real line.

OUTPUT:

Integer or infinity. The size of a discrete set is the number of points; the size of a real interval is Infinity.

EXAMPLES:

```
sage: RealSet([0, 0], [1, 1], [3, 3]).cardinality()
3
sage: RealSet(0,3).cardinality()
+Infinity
```

static closed (*lower*, *upper*)

Construct a closed interval

INPUT:

•`lower`, `upper` – two real numbers or infinity. They will be sorted if necessary.

OUTPUT:

A new `RealSet`.

EXAMPLES:

```
sage: RealSet.closed(1, 0)
[0, 1]
```

static closed_open (*lower*, *upper*)

Construct an half-open interval

INPUT:

•`lower`, `upper` – two real numbers or infinity. They will be sorted if necessary.

OUTPUT:

A new `RealSet` that is closed at the lower bound and open an the upper bound.

EXAMPLES:

```
sage: RealSet.closed_open(1, 0)
[0, 1)
```

complement ()

Return the complement

OUTPUT:

The set-theoretic complement as a new `RealSet`.

EXAMPLES:

```
sage: RealSet(0,1).complement()
(-oo, 0] + [1, +oo)

sage: s1 = RealSet(0,2) + RealSet.unbounded_above_closed(10);  s1
(0, 2) + [10, +oo)
sage: s1.complement()
(-oo, 0] + [2, 10)

sage: s2 = RealSet(1,3) + RealSet.unbounded_below_closed(-10);  s2
(-oo, -10] + (1, 3)
sage: s2.complement()
(-10, 1] + [3, +oo)
```

contains (x)

Return whether x is contained in the set

INPUT:

•`x` – a real number.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: s = RealSet(0,2) + RealSet.unbounded_above_closed(10);  s
(0, 2) + [10, +oo)
sage: s.contains(1)
True
sage: s.contains(0)
False
sage: 10 in s    # syntactic sugar
True
```

difference(**other*)

Return `self` with `other` subtracted

INPUT:

- `other` – a `RealSet` or data that defines one.

OUTPUT:

The set-theoretic difference of `self` with `other` removed as a new `RealSet`.

EXAMPLES:

```
sage: s1 = RealSet(0,2) + RealSet.unbounded_above_closed(10);  s1
(0, 2) + [10, +oo)
sage: s2 = RealSet(1,3) + RealSet.unbounded_below_closed(-10);  s2
(-oo, -10] + (1, 3)
sage: s1.difference(s2)
(0, 1] + [10, +oo)
sage: s1 - s2    # syntactic sugar
(0, 1] + [10, +oo)
sage: s2.difference(s1)
(-oo, -10] + [2, 3)
sage: s2 - s1    # syntactic sugar
(-oo, -10] + [2, 3)
sage: s1.difference(1,11)
(0, 1] + [11, +oo)
```

get_interval(*i*)

Return the `i`-th connected component.

Note that the intervals representing the real set are always normalized, see `normalize()`.

INPUT:

- `i` – integer.

OUTPUT:

The i-th connected component as a `RealInterval`.

EXAMPLES:

```
sage: s = RealSet(RealSet.open_closed(0,1), RealSet.closed_open(2,3))
sage: s.get_interval(0)
(0, 1]
sage: s[0]    # shorthand
(0, 1]
sage: s.get_interval(1)
[2, 3)
sage: s[0] == s.get_interval(0)
True
```

inf()

Return the infimum

OUTPUT:

A real number or infinity.

EXAMPLES:

```
sage: s1 = RealSet(0,2) + RealSet.unbounded_above_closed(10);  s1
(0, 2) + [10, +oo)
sage: s1.inf()
0

sage: s2 = RealSet(1,3) + RealSet.unbounded_below_closed(-10);  s2
(-oo, -10] + (1, 3)
sage: s2.inf()
-Infinity
```

intersection(**other*)

Return the intersection of the two sets

INPUT:

- `other` – a `RealSet` or data that defines one.

OUTPUT:

The set-theoretic intersection as a new `RealSet`.

EXAMPLES:

```
sage: s1 = RealSet(0,2) + RealSet.unbounded_above_closed(10);  s1
(0, 2) + [10, +oo)
sage: s2 = RealSet(1,3) + RealSet.unbounded_below_closed(-10);  s2
(-oo, -10] + (1, 3)
sage: s1.intersection(s2)
(1, 2)
sage: s1 & s2    # syntactic sugar
(1, 2)

sage: s1 = RealSet((0, 1), (2, 3));  s1
(0, 1) + (2, 3)
sage: s2 = RealSet([0, 1], [2, 3]);  s2
[0, 1] + [2, 3]
sage: s3 = RealSet([1, 2]);  s3
[1, 2]
sage: s1.intersection(s2)
(0, 1) + (2, 3)
sage: s1.intersection(s3)
{}
sage: s2.intersection(s3)
{1} + {2}
```

is_disjoint_from(**other*)

Test whether the two sets are disjoint

INPUT:

- `other` – a `RealSet` or data defining one.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: s1 = RealSet((0, 1), (2, 3));  s1
(0, 1) + (2, 3)
sage: s2 = RealSet([1, 2]);  s2
[1, 2]
sage: s1.is_disjoint_from(s2)
True
sage: s1.is_disjoint_from([1, 2])
True
```

is_empty()

Return whether the set is empty

EXAMPLES:

```
sage: RealSet(0, 1).is_empty()
False
sage: RealSet(0, 0).is_empty()
True
```

is_included_in(**other*)

Tests interval inclusion

INPUT:

- `*args` – a `RealSet` or something that defines one.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: I = RealSet((1,2))
sage: J = RealSet((1,3))
sage: K = RealSet((2,3))
sage: I.is_included_in(J)
True
sage: J.is_included_in(K)
False
```

n_components()

Return the number of connected components

See also `get_interval()`

EXAMPLES:

```
sage: s = RealSet(RealSet.open_closed(0,1), RealSet.closed_open(2,3))
sage: s.n_components()
2
```

static normalize(*intervals*)

Bring a collection of intervals into canonical form

INPUT:

- `intervals` – a list/tuple/iterable of intervals.

OUTPUT:

A tuple of intervals such that

- they are sorted in ascending order (by lower bound)

- there is a gap between each interval
- all intervals are non-empty

EXAMPLES:

```
sage: i1 = RealSet((0, 1))[0]
sage: i2 = RealSet([1, 2])[0]
sage: i3 = RealSet((2, 3))[0]
sage: RealSet.normalize([i1, i2, i3])
((0, 3),)

sage: RealSet((0, 1), [1, 2], (2, 3))
(0, 3)
sage: RealSet((0, 1), (1, 2), (2, 3))
(0, 1) + (1, 2) + (2, 3)
sage: RealSet([0, 1], [2, 3])
[0, 1] + [2, 3]
sage: RealSet((0, 2), (1, 3))
(0, 3)
sage: RealSet(0,0)
{}
```

static open (*lower*, *upper*)

Construct an open interval

INPUT:

- `lower`, `upper` – two real numbers or infinity. They will be sorted if necessary.

OUTPUT:

A new `RealSet`.

EXAMPLES:

```
sage: RealSet.open(1, 0)
(0, 1)
```

static open_closed (*lower*, *upper*)

Construct a half-open interval

INPUT:

- `lower`, `upper` – two real numbers or infinity. They will be sorted if necessary.

OUTPUT:

A new `RealSet` that is open at the lower bound and closed at the upper bound.

EXAMPLES:

```
sage: RealSet.open_closed(1, 0)
(0, 1]
```

static point (*p*)

Construct an interval containing a single point

INPUT:

- `p` – a real number.

OUTPUT:

A new `RealSet`.

EXAMPLES:

```
sage: RealSet.open(1, 0)
(0, 1)
```

sup()

Return the supremum

OUTPUT:

A real number or infinity.

EXAMPLES:

```
sage: s1 = RealSet(0,2) + RealSet.unbounded_above_closed(10);  s1
(0, 2) + [10, +oo)
sage: s1.sup()
+Infinity

sage: s2 = RealSet(1,3) + RealSet.unbounded_below_closed(-10);  s2
(-oo, -10] + (1, 3)
sage: s2.sup()
3
```

static **unbounded_above_closed**(*bound*)

Construct a semi-infinite interval

INPUT:

- `bound` – a real number.

OUTPUT:

A new `RealSet` from the bound (including) to plus infinity.

EXAMPLES:

```
sage: RealSet.unbounded_above_closed(1)
[1, +oo)
```

static **unbounded_above_open**(*bound*)

Construct a semi-infinite interval

INPUT:

- `bound` – a real number.

OUTPUT:

A new `RealSet` from the bound (excluding) to plus infinity.

EXAMPLES:

```
sage: RealSet.unbounded_above_open(1)
(1, +oo)
```

static **unbounded_below_closed**(*bound*)

Construct a semi-infinite interval

INPUT:

- `bound` – a real number.

OUTPUT:

A new `RealSet` from minus infinity to the bound (including).

EXAMPLES:

```
sage: RealSet.unbounded_below_closed(1)
(-oo, 1]
```

static unbounded_below_open (*bound*)

Construct a semi-infinite interval

INPUT:

- `bound` – a real number.

OUTPUT:

A new `RealSet` from minus infinity to the bound (excluding).

EXAMPLES:

```
sage: RealSet.unbounded_below_open(1)
(-oo, 1)
```

union (**other*)

Return the union of the two sets

INPUT:

- `other` – a `RealSet` or data that defines one.

OUTPUT:

The set-theoretic union as a new `RealSet`.

EXAMPLES:

```
sage: s1 = RealSet(0,2)
sage: s2 = RealSet(1,3)
sage: s1.union(s2)
(0, 3)
sage: s1.union(1,3)
(0, 3)
sage: s1 | s2    # syntactic sugar
(0, 3)
sage: s1 + s2    # syntactic sugar
(0, 3)
```

CHAPTER

THIRTYSIX

BASE CLASS FOR PARENT OBJECTS

CLASS HIERARCHY:

```
SageObject
    CategoryObject
        Parent
```

A simple example of registering coercions:

```
sage: class A_class(Parent):
....:   def __init__(self, name):
....:       Parent.__init__(self, name=name)
....:       self._populate_coercion_lists_()
....:       self.rename(name)
....:   #
....:   def category(self):
....:       return Sets()
....:   #
....:   def _element_constructor_(self, i):
....:       assert(isinstance(i, (int, Integer)))
....:       return ElementWrapper(self, i)
....:
sage: A = A_class("A")
sage: B = A_class("B")
sage: C = A_class("C")

sage: def f(a):
....:   return B(a.value+1)
....:
sage: class MyMorphism(Morphism):
....:   def __init__(self, domain, codomain):
....:       Morphism.__init__(self, Hom(domain, codomain))
....:   #
....:   def _call_(self, x):
....:       return self.codomain()(x.value)
....:
sage: f = MyMorphism(A,B)
sage: f
    Generic morphism:
      From: A
      To:   B
sage: B.register_coercion(f)
sage: C.register_coercion(MyMorphism(B,C))
sage: A(A(1)) == A(1)
True
sage: B(A(1)) == B(1)
```

```
True
sage: C(A(1)) == C(1)
True

sage: A(B(1))
Traceback (most recent call last):
...
AssertionError
```

When implementing an element of a ring, one would typically provide the element class with `_rmul_` and/or `_lmul_` methods for the action of a base ring, and with `_mul_` for the ring multiplication. However, prior to trac ticket #14249, it would have been necessary to additionally define a method `_an_element_()` for the parent. But now, the following example works:

```
sage: from sage.structure.element import RingElement
sage: class MyElement(RingElement):
....:     def __init__(self, parent, x, y):
....:         RingElement.__init__(self, parent)
....:     def _mul_(self, other):
....:         return self
....:     def _rmul_(self, other):
....:         return self
....:     def _lmul_(self, other):
....:         return self
sage: class MyParent(Parent):
....:     Element = MyElement
```

Now, we define

```
sage: P = MyParent(base=ZZ, category=Rings())
sage: a = P(1,2)
sage: a*a is a
True
sage: a*2 is a
True
sage: 2*a is a
True
```

TESTS:

This came up in some subtle bug once:

```
sage: gp(2) + gap(3)
5
```

class `sage.structure.parent.`**`EltPair`**

Bases: `object`

`short_repr`()

class `sage.structure.parent.`**`Parent`**

Bases: `sage.structure.category_object.CategoryObject`

Base class for all parents.

Parents are the Sage/mathematical analogues of container objects in computer science.

INPUT:

- `base` – An algebraic structure considered to be the "base" of this parent (e.g. the base field for a vector space).
- `category` – a category or list/tuple of categories. The category in which this parent lies (or list or tuple thereof). Since categories support more general super-categories, this should be the most specific category possible. If category is a list or tuple, a JoinCategory is created out of them. If category is not specified, the category will be guessed (see `CategoryObject`), but won't be used to inherit parent's or element's code from this category.
- `element_constructor` – A class or function that creates elements of this Parent given appropriate input (can also be filled in later with `_populate_coercion_lists_()`)
- `gens` – Generators for this object (can also be filled in later with `_populate_generators_()`)
- `names` – Names of generators.
- `normalize` – Whether to standardize the names (remove punctuation, etc)
- `facade` – a parent, or tuple thereof, or `True`

If `facade` is specified, then `Sets().Facade()` is added to the categories of the parent. Furthermore, if `facade` is not `True`, the internal attribute `_facade_for` is set accordingly for use by `Sets.Facade.ParentMethods.facade_for()`.

Internal invariants:

- `self._element_init_pass_parent == guess_pass_parent(self, self._element_constructor)` Ensures that `__call__()` passes down the parent properly to `_element_constructor()`. See trac ticket #5979.

Todo

Eventually, category should be `Sets` by default.

TESTS:

We check that the facade option is compatible with specifying categories as a tuple:

```
sage: class MyClass(Parent): pass
sage: P = MyClass(facade = ZZ, category = (Monoids(), CommutativeAdditiveMonoids()))
sage: P.category()
Join of Category of monoids and Category of commutative additive monoids and Category of facade
```

`__call__`(*x=0*, **args*, ***kwds*)

This is the generic call method for all parents.

When called, it will find a map based on the Parent (or type) of x. If a coercion exists, it will always be chosen. This map will then be called (with the arguments and keywords if any).

By default this will dispatch as quickly as possible to `_element_constructor_()` though faster pathways are possible if so desired.

TESTS:

We check that the invariant:

```
self._element_init_pass_parent == guess_pass_parent(self, self._element_constructor)
```

is preserved (see trac ticket #5979):

```
sage: class MyParent(Parent):
....:     def _element_constructor_(self, x):
....:         print self, x
....:         return sage.structure.element.Element(parent = self)
```

```
....:     def _repr_(self):
....:         return "my_parent"
....:
sage: my_parent = MyParent()
sage: x = my_parent("bla")
my_parent bla
sage: x.parent()         # indirect doctest
my_parent

sage: x = my_parent()    # shouldn't this one raise an error?
my_parent 0
sage: x = my_parent(3)   # todo: not implemented  why does this one fail???
my_parent 3
```

_populate_coercion_lists_(*coerce_list=[]*, *action_list=[]*, *convert_list=[]*, *embedding=None*, *convert_method_name=None*, *element_constructor=None*, *init_no_parent=None*, *unpickling=False*)

This function allows one to specify coercions, actions, conversions and embeddings involving this parent.

IT SHOULD ONLY BE CALLED DURING THE __INIT__ method, often at the end.

INPUT:

- `coerce_list` – a list of coercion Morphisms to self and parents with canonical coercions to self
- `action_list` – a list of actions on and by self
- **`convert_list` – a list of conversion Maps to self and** parents with conversions to self
- `embedding` – a single Morphism from self
- `convert_method_name` – a name to look for that other elements can implement to create elements of self (e.g. _integer_)
- `element_constructor` – A callable object used by the __call__ method to construct new elements. Typically the element class or a bound method (defaults to self._element_constructor_).
- `init_no_parent` – if True omit passing self in as the first argument of element_constructor for conversion. This is useful if parents are unique, or element_constructor is a bound method (this latter case can be detected automatically).

__mul__(*x*)

This is a multiplication method that more or less directly calls another attribute `_mul_` (single underscore). This is because `__mul__` can not be implemented via inheritance from the parent methods of the category, but `_mul_` can be inherited. This is, e.g., used when creating twosided ideals of matrix algebras. See trac ticket #7797.

EXAMPLE:

```
sage: MS = MatrixSpace(QQ,2,2)
```

This matrix space is in fact an algebra, and in particular it is a ring, from the point of view of categories:

```
sage: MS.category()
Category of algebras over quotient fields
sage: MS in Rings()
True
```

However, its class does not inherit from the base class `Ring`:

```
sage: isinstance(MS,Ring)
False
```

Its `_mul_` method is inherited from the category, and can be used to create a left or right ideal:

```
sage: MS._mul_.__module__
'sage.categories.rings'
sage: MS*MS.1      # indirect doctest
Left Ideal
(
  [0 1]
  [0 0]
)
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: MS*[MS.1,2]
Left Ideal
(
  [0 1]
  [0 0],

  [2 0]
  [0 2]
)
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: MS.1*MS
Right Ideal
(
  [0 1]
  [0 0]
)
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: [MS.1,2]*MS
Right Ideal
(
  [0 1]
  [0 0],

  [2 0]
  [0 2]
)
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

`__contains__`(*x*)

True if there is an element of self that is equal to x under ==, or if x is already an element of self. Also, True in other cases involving the Symbolic Ring, which is handled specially.

For many structures we test this by using `__call__()` and then testing equality between x and the result.

The Symbolic Ring is treated differently because it is ultra-permissive about letting other rings coerce in, but ultra-strict about doing comparisons.

EXAMPLES:

```
sage: 2 in Integers(7)
True
sage: 2 in ZZ
True
sage: Integers(7)(3) in ZZ
True
sage: 3/1 in ZZ
```

```
True
sage: 5 in QQ
True
sage: I in RR
False
sage: SR(2) in ZZ
True
sage: RIF(1, 2) in RIF
True
sage: pi in RIF # there is no element of RIF equal to pi
False
sage: sqrt(2) in CC
True
sage: pi in RR
True
sage: pi in CC
True
sage: pi in RDF
True
sage: pi in CDF
True
```

TESTS:

Check that trac ticket #13824 is fixed:

```
sage: 4/3 in GF(3)
False
sage: 15/50 in GF(25, 'a')
False
sage: 7/4 in Integers(4)
False
sage: 15/36 in Integers(6)
False
```

_coerce_map_from_(S)

Override this method to specify coercions beyond those specified in coerce_list.

If no such coercion exists, return None or False. Otherwise, it may return either an actual Map to use for the coercion, a callable (in which case it will be wrapped in a Map), or True (in which case a generic map will be provided).

_convert_map_from_(S)

Override this method to provide additional conversions beyond those given in convert_list.

This function is called after coercions are attempted. If there is a coercion morphism in the opposite direction, one should consider adding a section method to that.

This MUST return a Map from S to self, or None. If None is returned then a generic map will be provided.

_get_action_(S, *op*, *self_on_left*)

Override this method to provide an action of self on S or S on self beyond what was specified in action_list.

This must return an action which accepts an element of self and an element of S (in the order specified by self_on_left).

_an_element_()

Returns an element of self. Want it in sufficient generality that poorly-written functions won't work when they're not supposed to. This is cached so doesn't have to be super fast.

EXAMPLES:

```
sage: QQ._an_element_()
1/2
sage: ZZ['x,y,z']._an_element_()
x
```

TESTS:

Since `Parent` comes before the parent classes provided by categories in the hierarchy of classes, we make sure that this default implementation of `_an_element_()` does not override some provided by the categories. Eventually, this default implementation should be moved into the categories to avoid this workaround:

```
sage: S = FiniteEnumeratedSet([1,2,3])
sage: S.category()
Category of facade finite enumerated sets
sage: super(Parent, S)._an_element_
Cached version of <function _an_element_from_iterator at ...>
sage: S._an_element_()
1
sage: S = FiniteEnumeratedSet([])
sage: S._an_element_()
Traceback (most recent call last):
...
EmptySetError
```

_repr_option(*key*)

Metadata about the `_repr_()` output.

INPUT:

- `key` – string. A key for different metadata informations that can be inquired about.

Valid `key` arguments are:

- `'ascii_art'`: The `_repr_()` output is multi-line ascii art and each line must be printed starting at the same column, or the meaning is lost.
- `'element_ascii_art'`: same but for the output of the elements. Used in `sage.repl.display.formatter`.
- `'element_is_atomic'`: the elements print atomically, that is, parenthesis are not required when *printing* out any of $x - y$, $x + y$, x^y and x/y.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: ZZ._repr_option('ascii_art')
False
sage: MatrixSpace(ZZ, 2)._repr_option('element_ascii_art')
True
```

_init_category_(*category*)

Initialize the category framework

Most parents initialize their category upon construction, and this is the recommended behavior. For example, this happens when the constructor calls `Parent.__init__()` directly or indirectly. However, some parents defer this for performance reasons. For example, `sage.matrix.matrix_space.MatrixSpace` does not.

EXAMPLES:

```
sage: P = Parent()
sage: P.category()
Category of sets
sage: class MyParent(Parent):
....:     def __init__(self):
....:         self._init_category_(Groups())
sage: MyParent().category()
Category of groups
```

Hom(*codomain*, *category=None*)

Return the homspace `Hom(self, codomain, category)`.

INPUT:

- `codomain` – a parent
- `category` – a category or `None` (default: `None`) If `None`, the meet of the category of `self` and `codomain` is used.

OUTPUT:

The homspace of all homomorphisms from `self` to `codomain` in the category `category`.

See also:

`Hom()`

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: R.Hom(QQ)
Set of Homomorphisms from Multivariate Polynomial Ring in x, y over Rational Field to Ration
```

Homspaces are defined for very general Sage objects, even elements of familiar rings:

```
sage: n = 5; Hom(n,7)
Set of Morphisms from 5 to 7 in Category of elements of Integer Ring
sage: z=(2/3); Hom(z,8/1)
Set of Morphisms from 2/3 to 8 in Category of elements of Rational Field
```

This example illustrates the optional third argument:

```
sage: QQ.Hom(ZZ, Sets())
Set of Morphisms from Rational Field to Integer Ring in Category of sets
```

A parent may specify how to construct certain homsets by implementing a method _Hom_`(codomain, category). See :func:`~sage.categories.homset.Hom()` for details.

an_element()

Returns a (preferably typical) element of this parent.

This is used both for illustration and testing purposes. If the set `self` is empty, `an_element()` raises the exception `EmptySetError`.

This calls `_an_element_()` (which see), and caches the result. Parent are thus encouraged to override `_an_element_()`.

EXAMPLES:

```
sage: CDF.an_element()
1.0*I
sage: ZZ[['t']].an_element()
t
```

In case the set is empty, an EmptySetError is raised:

```
sage: Set([]).an_element()
Traceback (most recent call last):
...
EmptySetError
```

category()

EXAMPLES:

```
sage: P = Parent()
sage: P.category()
Category of sets
sage: class MyParent(Parent):
....:     def __init__(self): pass
sage: MyParent().category()
Category of sets
```

coerce(x)

Return x as an element of self, if and only if there is a canonical coercion from the parent of x to self.

EXAMPLES:

```
sage: QQ.coerce(ZZ(2))
2
sage: ZZ.coerce(QQ(2))
Traceback (most recent call last):
...
TypeError: no canonical coercion from Rational Field to Integer Ring
```

We make an exception for zero:

```
sage: V = GF(7)^7
sage: V.coerce(0)
(0, 0, 0, 0, 0, 0, 0)
```

coerce_embedding()

Return the embedding of self into some other parent, if such a parent exists.

This does not mean that there are no coercion maps from self into other fields, this is simply a specific morphism specified out of self and usually denotes a special relationship (e.g. sub-objects, choice of completion, etc.)

EXAMPLES:

```
sage: K.<a>=NumberField(x^3+x^2+1,embedding=1)
sage: K.coerce_embedding()
Generic morphism:
  From: Number Field in a with defining polynomial x^3 + x^2 + 1
  To:   Real Lazy Field
  Defn: a -> -1.465571231876768?
sage: K.<a>=NumberField(x^3+x^2+1,embedding=CC.gen())
sage: K.coerce_embedding()
Generic morphism:
  From: Number Field in a with defining polynomial x^3 + x^2 + 1
  To:   Complex Lazy Field
  Defn: a -> 0.2327856159383841? + 0.7925519925154479?*I
```

coerce_map_from(S)

Return a `Map` object to coerce from `S` to `self` if one exists, or `None` if no such coercion exists.

EXAMPLES:

By trac ticket #12313, a special kind of weak key dictionary is used to store coercion and conversion maps, namely `MonoDict`. In that way, a memory leak was fixed that would occur in the following test:

```
sage: import gc
sage: _ = gc.collect()
sage: K = GF(1<<55,'t')
sage: for i in range(50):
....:     a = K.random_element()
....:     E = EllipticCurve(j=a)
....:     b = K.has_coerce_map_from(E)
sage: _ = gc.collect()
sage: len([x for x in gc.get_objects() if isinstance(x,type(E))])
1
```

TESTS:

The following was fixed in trac ticket #12969:

```
sage: R = QQ['q,t'].fraction_field()
sage: Sym = sage.combinat.sf.sf.SymmetricFunctions(R)
sage: H = Sym.macdonald().H()
sage: P = Sym.macdonald().P()
sage: m = Sym.monomial()
sage: Ht = Sym.macdonald().Ht()
sage: phi = m.coerce_map_from(P)
```

construction()

Returns a pair (functor, parent) such that functor(parent) return self. If this ring does not have a functorial construction, return None.

EXAMPLES:

```
sage: QQ.construction()
(FractionField, Integer Ring)
sage: f, R = QQ['x'].construction()
sage: f
Poly[x]
sage: R
Rational Field
sage: f(R)
Univariate Polynomial Ring in x over Rational Field
```

convert_map_from(S)

This function returns a `Map` from S to $self$, which may or may not succeed on all inputs. If a coercion map from S to self exists, then the it will be returned. If a coercion from $self$ to S exists, then it will attempt to return a section of that map.

Under the new coercion model, this is the fastest way to convert elements of S to elements of $self$ (short of manually constructing the elements) and is used by `__call__()`.

EXAMPLES:

```
sage: m = ZZ.convert_map_from(QQ)
sage: m
Generic map:
  From: Rational Field
  To:   Integer Ring
sage: m(-35/7)
```

```
-5
sage: parent(m(-35/7))
Integer Ring
```

element_class()

The (default) class for the elements of this parent

FIXME's and design issues:

- If self.Element is "trivial enough", should we optimize it away with: self.element_class = dynamic_class("%s.element_class"%self.__class__.__name__, (category.element_class,), self.Element)
- This should lookup for Element classes in all super classes

get_action(*S*, *op=None*, *self_on_left=True*, *self_el=None*, *S_el=None*)

Returns an action of self on S or S on self.

To provide additional actions, override `_get_action_()`.

TESTS:

```
sage: M = QQ['y']^3
sage: M.get_action(ZZ['x']['y'])
Right scalar multiplication by Univariate Polynomial Ring in y over Univariate Polynomial Ri
sage: M.get_action(ZZ['x']) # should be None
```

has_coerce_map_from(*S*)

Return True if there is a natural map from S to self. Otherwise, return False.

EXAMPLES:

```
sage: RDF.has_coerce_map_from(QQ)
True
sage: RDF.has_coerce_map_from(QQ['x'])
False
sage: RDF['x'].has_coerce_map_from(QQ['x'])
True
sage: RDF['x,y'].has_coerce_map_from(QQ['x'])
True
```

hom(*im_gens*, *codomain=None*, *check=None*)

Return the unique homomorphism from self to codomain that sends `self.gens()` to the entries of `im_gens`. Raises a TypeError if there is no such homomorphism.

INPUT:

- `im_gens` – the images in the codomain of the generators of this object under the homomorphism
- `codomain` – the codomain of the homomorphism
- `check` – whether to verify that the images of generators extend to define a map (using only canonical coercions).

OUTPUT:

A homomorphism self –> codomain

Note: As a shortcut, one can also give an object X instead of `im_gens`, in which case return the (if it exists) natural map to X.

EXAMPLES:

Polynomial Ring: We first illustrate construction of a few homomorphisms involving a polynomial ring:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = R.hom([5], QQ)
sage: f(x^2 - 19)
6

sage: R.<x> = PolynomialRing(QQ)
sage: f = R.hom([5], GF(7))
Traceback (most recent call last):
...
TypeError: images do not define a valid homomorphism

sage: R.<x> = PolynomialRing(GF(7))
sage: f = R.hom([3], GF(49,'a'))
sage: f
Ring morphism:
  From: Univariate Polynomial Ring in x over Finite Field of size 7
  To:   Finite Field in a of size 7^2
  Defn: x |--> 3
sage: f(x+6)
2
sage: f(x^2+1)
3
```

Natural morphism:

```
sage: f = ZZ.hom(GF(5))
sage: f(7)
2
sage: f
Ring Coercion morphism:
  From: Integer Ring
  To:   Finite Field of size 5
```

There might not be a natural morphism, in which case a `TypeError` is raised:

```
sage: QQ.hom(ZZ)
Traceback (most recent call last):
...
TypeError: Natural coercion morphism from Rational Field to Integer Ring not defined.
```

is_atomic_repr()

The old way to signal atomic string reps.

True if the elements have atomic string representations, in the sense that if they print at s, then -s means the negative of s. For example, integers are atomic but polynomials are not.

EXAMPLES:

```
sage: Parent().is_atomic_repr()
doctest:...: DeprecationWarning: Use _repr_option to return metadata about string rep
See http://trac.sagemath.org/14040 for details.
False
```

is_coercion_cached(*domain*)

is_conversion_cached(*domain*)

is_exact()

Test whether the ring is exact.

Note: This defaults to true, so even if it does return `True` you have no guarantee (unless the ring has properly overloaded this).

OUTPUT:

Return True if elements of this ring are represented exactly, i.e., there is no precision loss when doing arithmetic.

EXAMPLES:

```
sage: QQ.is_exact()
True
sage: ZZ.is_exact()
True
sage: Qp(7).is_exact()
False
sage: Zp(7, type='capped-abs').is_exact()
False
```

register_action(*action*)

Update the coercion model to use `action` to act on self.

`action` should be of type `sage.categories.action.Action`.

EXAMPLES:

```
sage: import sage.categories.action
sage: import operator

sage: class SymmetricGroupAction(sage.categories.action.Action):
....:     "Act on a multivariate polynomial ring by permuting the generators."
....:     def __init__(self, G, M, is_left=True):
....:         sage.categories.action.Action.__init__(self, G, M, is_left, operator.mul)
....:
....:     def _call_(self, g, a):
....:         if not self.is_left():
....:             g, a = a, g
....:         D = {}
....:         for k, v in a.dict().items():
....:             nk = [0]*len(k)
....:             for i in range(len(k)):
....:                 nk[g(i+1)-1] = k[i]
....:             D[tuple(nk)] = v
....:         return a.parent()(D)

sage: R.<x, y, z> = QQ['x, y, z']
sage: G = SymmetricGroup(3)
sage: act = SymmetricGroupAction(G, R)
sage: t = x + 2*y + 3*z

sage: act(G((1, 2)), t)
2*x + y + 3*z
sage: act(G((2, 3)), t)
x + 3*y + 2*z
sage: act(G((1, 2, 3)), t)
3*x + y + 2*z
```

This should fail, since we haven't registered the left action:

```
sage: G((1,2)) * t
Traceback (most recent call last):
```

```
...
TypeError: ...
```

Now let's make it work:

```
sage: R._unset_coercions_used()
sage: R.register_action(act)
sage: G((1, 2)) * t
2*x + y + 3*z
```

register_coercion(*mor*)

Update the coercion model to use $mor : P \to$ self to coerce from a parent `P` into `self`.

For safety, an error is raised if another coercion has already been registered or discovered between `P` and `self`.

EXAMPLES:

```
sage: K.<a> = ZZ['a']
sage: L.<b> = ZZ['b']
sage: L_into_K = L.hom([-a]) # non-trivial automorphism
sage: K.register_coercion(L_into_K)

sage: K(0) + b
-a
sage: a + b
0
sage: K(b) # check that convert calls coerce first; normally this is just a
-a

sage: L(0) + a in K # this goes through the coercion mechanism of K
True
sage: L(a) in L # this still goes through the convert mechanism of L
True

sage: K.register_coercion(L_into_K)
Traceback (most recent call last):
...
AssertionError: coercion from Univariate Polynomial Ring in b over Integer Ring to Univariat
```

register_conversion(*mor*)

Update the coercion model to use mor : $P \to$ self to convert from `P` into `self`.

EXAMPLES:

```
sage: K.<a> = ZZ['a']
sage: M.<c> = ZZ['c']
sage: M_into_K = M.hom([a]) # trivial automorphism
sage: K._unset_coercions_used()
sage: K.register_conversion(M_into_K)

sage: K(c)
a
sage: K(0) + c
Traceback (most recent call last):
...
TypeError: ...
```

register_embedding(*embedding*)

Add embedding to coercion model.

This method updates the coercion model to use embedding : self $\to P$ to embed `self` into the parent `P`.

There can only be one embedding registered; it can only be registered once; and it must be registered before using this parent in the coercion model.

EXAMPLES:

```
sage: S3 = AlternatingGroup(3)
sage: G = SL(3, QQ)
sage: p = S3[2]; p.matrix()
[0 0 1]
[1 0 0]
[0 1 0]
```

In general one can't mix matrices and permutations:

```
sage: G(p)
Traceback (most recent call last):
...
TypeError: entries must be coercible to a list or integer
sage: phi = S3.hom(lambda p: G(p.matrix()), codomain = G)
sage: phi(p)
[0 0 1]
[1 0 0]
[0 1 0]
sage: S3._unset_coercions_used()
sage: S3.register_embedding(phi)
```

By trac ticket #14711, coerce maps should be copied when using outside of the coercion system:

```
sage: phi = copy(S3.coerce_embedding()); phi
Generic morphism:
  From: Alternating group of order 3!/2 as a permutation group
  To:   Special Linear Group of degree 3 over Rational Field
sage: phi(p)
[0 0 1]
[1 0 0]
[0 1 0]
```

This does not work since matrix groups are still old-style parents (see trac ticket #14014):

```
sage: G(p)                               # todo: not implemented
```

Though one can have a permutation act on the rows of a matrix:

```
sage: G(1) * p
[0 0 1]
[1 0 0]
[0 1 0]
```

Some more advanced examples:

```
sage: x = QQ['x'].0
sage: t = abs(ZZ.random_element(10^6))
sage: K = NumberField(x^2 + 2*3*7*11, "a"+str(t))
sage: a = K.gen()
sage: K_into_MS = K.hom([a.matrix()])
sage: K._unset_coercions_used()
sage: K.register_embedding(K_into_MS)

sage: L = NumberField(x^2 + 2*3*7*11*19*31, "b"+str(abs(ZZ.random_element(10^6))))
sage: b = L.gen()
```

```
sage: L_into_MS = L.hom([b.matrix()])
sage: L._unset_coercions_used()
sage: L.register_embedding(L_into_MS)

sage: K.coerce_embedding()(a)
[   0    1]
[-462    0]
sage: L.coerce_embedding()(b)
[      0       1]
[-272118       0]

sage: a.matrix() * b
[-272118       0]
[      0    -462]
sage: a * b.matrix()
[-272118       0]
[      0    -462]
```

`sage.structure.parent.`**`Set_PythonType`**`(`*theType*`)`

Return the (unique) Parent that represents the set of Python objects of a specified type.

EXAMPLES:

```
sage: from sage.structure.parent import Set_PythonType
sage: Set_PythonType(list)
Set of Python objects of type 'list'
sage: Set_PythonType(list) is Set_PythonType(list)
True
sage: S = Set_PythonType(tuple)
sage: S([1,2,3])
(1, 2, 3)
```

```
S is a parent which models the set of all lists:
  sage: S.category()
  Category of sets
```

EXAMPLES:

```
sage: R = sage.structure.parent.Set_PythonType(int)
sage: S = sage.structure.parent.Set_PythonType(float)
sage: Hom(R, S)
Set of Morphisms from Set of Python objects of type 'int' to Set of Python objects of type 'floa
```

class `sage.structure.parent.`**`Set_PythonType_class`**

Bases: `sage.structure.parent.Set_generic`

The set of Python objects of a given type.

EXAMPLES:

```
sage: S = sage.structure.parent.Set_PythonType(int)
sage: S
Set of Python objects of type 'int'
sage: int('1') in S
True
sage: Integer('1') in S
False

sage: sage.structure.parent.Set_PythonType(2)
Traceback (most recent call last):
```

```
...
TypeError: must be intialized with a type, not 2
```

cardinality()

EXAMPLES:

```
sage: S = sage.structure.parent.Set_PythonType(bool)
sage: S.cardinality()
2
sage: S = sage.structure.parent.Set_PythonType(int)
sage: S.cardinality()
4294967296                        # 32-bit
18446744073709551616              # 64-bit
sage: S = sage.structure.parent.Set_PythonType(float)
sage: S.cardinality()
18437736874454810627
sage: S = sage.structure.parent.Set_PythonType(long)
sage: S.cardinality()
+Infinity
```

object()

EXAMPLES:

```
sage: S = sage.structure.parent.Set_PythonType(tuple)
sage: S.object()
<type 'tuple'>
```

class sage.structure.parent.**Set_generic**

Bases: sage.structure.parent.Parent

Abstract base class for sets.

TESTS:

```
sage: Set(QQ).category()
Category of sets
```

object()

sage.structure.parent.**is_Parent**(*x*)

Return True if x is a parent object, i.e., derives from sage.structure.parent.Parent and False otherwise.

EXAMPLES:

```
sage: from sage.structure.parent import is_Parent
sage: is_Parent(2/3)
False
sage: is_Parent(ZZ)
True
sage: is_Parent(Primes())
True
```

sage.structure.parent.**normalize_names**(*ngens*, *names*)

TESTS:

```
sage: sage.structure.parent.normalize_names(5, 'x')
('x0', 'x1', 'x2', 'x3', 'x4')
sage: sage.structure.parent.normalize_names(2, ['x','y'])
('x', 'y')
```

CHAPTER

THIRTYSEVEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

[Pro] Prototype pattern http://en.wikipedia.org/wiki/Prototype_pattern

[GOF] Design Patterns: Elements of Reusable Object-Oriented Software. E. Gamma; R. Helm; R. Johnson; J. Vlissides (1994). Addison-Wesley. ISBN 0-201-63361-2.

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

Z