Sage Reference Manual: Miscellaneous

Release 6.6

The Sage Development Team

April 18, 2015

CONTENTS

CHAPTER

ONE

ABSTRACT METHODS

class sage.misc.abstract_method.**AbstractMethod**(*f*, *optional=False*)

Bases: object

Constructor for abstract methods

EXAMPLES:

```
sage: def f(x):
...       "doc of f"
...       return 1
...
sage: x = abstract_method(f); x
<abstract method f at ...>
sage: x.__doc__
'doc of f'
sage: x.__name__
'f'
sage: x.__module__
'__main__'
```

is_optional()

Returns whether an abstract method is optional or not.

EXAMPLES:

```
sage: class AbstractClass:
...       @abstract_method
...       def required(): pass
...
...       @abstract_method(optional = True)
...       def optional(): pass
sage: AbstractClass.required.is_optional()
False
sage: AbstractClass.optional.is_optional()
True
```

sage.misc.abstract_method.**abstract_method**(*f=None*, *optional=False*)

Abstract methods

INPUT:

- f – a function
- optional – a boolean; defaults to False

The decorator abstract_method can be used to declare methods that should be implemented by all concrete derived classes. This declaration should typically include documentation for the specification for this method.

The purpose is to enforce a consistent and visual syntax for such declarations. It is used by the Sage categories for automated tests (see `Sets.Parent.test_not_implemented`).

EXAMPLES:

We create a class with an abstract method:

```
sage: class A(object):
...
...       @abstract_method
...       def my_method(self):
...           '''
...           The method :meth:`my_method` computes my_method
...
...           EXAMPLES::
...
...           '''
...           pass
...
sage: A.my_method
<abstract method my_method at ...>
```

The current policy is that a `NotImplementedError` is raised when accessing the method through an instance, even before the method is called:

```
sage: x = A()
sage: x.my_method
Traceback (most recent call last):
...
NotImplementedError: <abstract method my_method at ...>
```

It is also possible to mark abstract methods as optional:

```
sage: class A(object):
...
...       @abstract_method(optional = True)
...       def my_method(self):
...           '''
...           The method :meth:`my_method` computes my_method
...
...           EXAMPLES::
...
...           '''
...           pass
...

sage: A.my_method
<optional abstract method my_method at ...>

sage: x = A()
sage: x.my_method
NotImplemented
```

The official mantra for testing whether an optional abstract method is implemented is:

```
# Fixme: sage -t complains about indentation below
#    sage: if x.my_method is not NotImplemented:
#    ...         x.my_method()
#    ...     else:
#    ...         print "x.my_method not available. Let's use some other trick."
#    ...
```

```
#    x.my_method not available. Let's use some other trick.
```

Discussion

The policy details are not yet fixed. The purpose of this first implementation is to let developers experiment with it and give feedback on what's most practical.

The advantage of the current policy is that attempts at using a non implemented methods are caught as early as possible. On the other hand, one cannot use introspection directly to fetch the documentation:

```
sage: x.my_method?      # todo: not implemented
```

Instead one needs to do:

```
sage: A._my_method?     # todo: not implemented
```

This could probably be fixed in `sage.misc.sageinspect`.

TODO: what should be the recommended mantra for existence testing from the class?

TODO: should extra information appear in the output? The name of the class? That of the super class where the abstract method is defined?

TODO: look for similar decorators on the web, and merge

Implementation details

Technically, an abstract_method is a non-data descriptor (see Invoking Descriptors in the Python reference manual).

The syntax `@abstract_method` w.r.t. @abstract_method(optional = True) is achieved by a little trick which we test here:

```
sage: abstract_method(optional = True)
<function <lambda> at ...>
sage: abstract_method(optional = True)(banner)
<optional abstract method banner at ...>
sage: abstract_method(banner, optional = True)
<optional abstract method banner at ...>
```

`sage.misc.abstract_method.`**`abstract_methods_of_class`**`(`*cls*`)`

Returns the required and optional abstract methods of the class

EXAMPLES:

```
sage: class AbstractClass:
...       @abstract_method
...       def required1(): pass
...
...       @abstract_method(optional = True)
...       def optional2(): pass
...
...       @abstract_method(optional = True)
...       def optional1(): pass
...
...       @abstract_method
...       def required2(): pass
...
```

```
sage: sage.misc.abstract_method.abstract_methods_of_class(AbstractClass)
{'optional': ['optional1', 'optional2'],
 'required': ['required1', 'required2']}
```

CHAPTER

TWO

ASCII ART

This file contains:

- `AsciiArt` an simple implementation of an ASCII art object,
- `ascii_art()` a function to get the ASCII art representation of any object in Sage,
- several others functions use to get ASCII art representation of primitive python elements (list, tuple, dict, set).

AUTHOR:

- Jean-Baptiste Priez (2013-04): initial version

EXAMPLES:

```
sage: n = var('n')
sage: integrate(n^2/x,x)
n^2*log(x)
sage: ascii_art(integrate(n^2/x,x))
 2
n *log(x)
sage: ascii_art(integrate(n^2/(pi*x),x))
 2
n *log(x)
---------
    pi
sage: ascii_art(list(Partitions(6)))
[                                                          * ]
[                                                      **  * ]
[                                       ***       **   *   * ]
[                     ****        ***   *    **   **   *   * ]
[         *****  ****  *     ***  **    *    **   *    *   * ]
[ ******, *    , **  , *   , ***, *   , *  , **, * , * , * ]
```

This method `ascii_art()` could be automatically use by the display hook manager activated by the magic function: `%display ascii_art`:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('%display ascii_art')
sage: shell.run_cell("i = var('i')")
sage: shell.run_cell('sum(factorial(i)*x^i, i, 0, 10)')
         10             9           8          7         6         5       4      3
3628800*x   + 362880*x  + 40320*x  + 5040*x  + 720*x  + 120*x  + 24*x  + 6*x

     2
+ 2*x  + x + 1
sage: shell.run_cell('3/(7*x)')
```

```
 3
---
7*x
sage: shell.run_cell('list(Compositions(5))')
[ *
[ *  **   *        *                   *
[ *  *   **  ***   *   **    *         *   **    *          *
[ *  *   *   *    **  **   ***  ****   *   *    **   ***    *    **     *
[ *, * , * , *  , * , *  , *  , *   , **, ** , ** , **  , ***, *** , ****,

      ]
      ]
      ]
      ]
***** ]
sage: shell.run_cell('%display simple')
sage: shell.quit()

                            .      ,    ,              ...
                            .?  .~?$NNO.II7.        ..GOGG
                             ., ~7NNI7NDG$       ...~~~MGG
              ..:IG..       ...G7:DDGNNDO..   ...7:~~~~GNN
              .~~GGGGG...   .O .=$+OD7GI$ ...GG:~+~~~~MG?M.7?
              .~~~D~~GGGGDGOM$..~+IN=NDG.G::D~~~?~~~7?NDI:???G
              .~~~~G:~~G~D:~NONGMMOGD$ND~~::I77~~$?++MN7G GI?D.
               7+I~~DG~~~N:GNGGOD$7OIOOMII::~:I7?G+OGN7G $III?.
                .7==I~~~++IN?+++?$7ND$==$G:G??????N$OG$  ~III?7
                 .$$+++?GG=+G?GDM?GOG:NGMGN7??????D$D~GODIII???.
                   D++++NG+DD$+=GGONGI$DNGDN??????,IN=DOGODN???.
                 , +~+++N??D$I+I=GNMDGDGNIINN??D+:::O~$GGGDIG?7
                :DD.:::OG?G=I~~G7GNM7777NIGODDNNG:::I$GGGGG7??O.
               ~GDDM:::GGMG+?+~$NNN$7IG7NMNMMMN, =::I7GGGGGO???~
              :GDND7?::OOM.D+O~GGDI77777DMGNM$.  ~:,$IGGGGGO???DO.
             OGDDNN.D77OO. $?7==GG777$7GNGMGO.     NOIGGGGGO???G$.
           .OODNNN,DIGDM$GGGGG==GGGGIGNDMDMG,      IGIGGGDON???GG:
          .GODNDM.G$I$IOIOMG$G$?GGGIO,7OD7GG.     ,GDIGGG??????GGG.
         .DGDNI7I7MDI+OOODN$$O,$7DMIN,,IOO77O.   G?$DIGGG?ID???OGG
         GGDNNMMO7GD+OOOGIMOG7::NN====:?MMNGIDD,..IINIGGG?I??DIOGG?
       .7ODMMMN.G7IOGOOODIMG,,:::$=~==::7OGG~IGOMDGMNIGGG????.OG$G.
       ?ODDMNNNO,II$OGODMGDMM?:DMG==~MDINNM$.7$IONDGI?GGG????:$GGG.
    .$MDMNNNNN..:?7GDDDGG,GGM?~:GGGDNND.GIM7D+GI$ON.:?GGG????$$OGG.
  .7DNDDDNMDOGG.=IGGND=7II+N??::$GIIO,IIGMG?7I7G$ON?,IGGG????7GIGG.
  ~GGGDNMMOOGGG$MGMMGDGMDGM?,G:GNG,:IIIGDG7IGGGGG$+NMIGGO?????IGGG.
 .GGGDDMM7OGNGMODMNDDDOO.MII?GI$7IIIIING7GGDMM.IGDG.G7GGGG??$?7GGG
.IGDDDDOINNGMGMDMNDDGDO...$OI+??7OIIIDDGN+$==I=GD ?,NGGGGN7????GGG
 7GGNDM$GONDGDD$MMGNDN. G.:$$G$?$7II$GOO,O=+7O7O~N?OM?GGGGGD+??$GG
 OOGDOI7DGMG..=~DG$DD.  $,,$$$D??7ODOOOODG$777$G OGMM:GGGGGG??GGG.
.$GDDG?7DMOGDNNMGGDGG ..,=O7$GG$+O$$OG=+O:GI77G$. ...DIGGGGI?$GGG.
.OGGGGO7OGDGGNGGGDGG.  O:,77$O$$$D     $GN:7777GD   ..$GGGG??GGG$.
.GGGGDI GD.~NOGDGGG    MG,777G77D7      +$~D77777=   ..7GGG???GGO.
  GGD.   .~NODN...   ..D::77O$7G77      .OG:G???G7.    7$NOM??GG=.
    .     . .       .+~~DD~77$G7OD7      .?GGO?$OM$D     ?????DGG...
                   7I77DN$II7$M.G$G      . .:G?==$G     .?????D??~
                 . .:IIIGO7O$GN..O$         7I=~+?I.    =????7? GGIG
                  $,III7NGGNNNG, .O.         O====7I.  .DG??$?. GG?G
                  .$7III$77777$$~  .          +====D$GG ~$???:  .$GIG.
                 :.+IIII$77$G$$O,+            OI==+7I$7=.:???     .GG$.
                 ,,III$I7?I7$$OG7D            ?+,==+77G?$,???.      GG?
```

```
         ?IIIII=+$I77777             .:++=+++O77:,??:         GG.
         .IIIIG+III77777.             7,=,++?II7$,?G          .GO  .
          $OII7?OIIIIIO              .I:++??IIIII???.           I$  .
          .DNGDMOIG777.                +~++?I$IIMN?,             .G,
           .G.I?IIDGII~                .OG??$$MIMGI.               G,
            N+?II7OGI7.                  DD?$GGDGID.                  :
             =?IIOOI$.                    ?~~GGG$II                  .+.
            ,7=IINGI                       ?=IGGGI?.
            G,+IGOII                        ?+II7??+
           .,:?IGOII                         ?+GIOII
         .:N+=?IMGI7=.                       .~:$I?IO.
       .$:IGO?$IIIII7+.                       O::I7I?.
      .:::=IIGIIIIIII.                        .~,$IG?IO
     .+:$IIIIMIII7G$?.                         I::I7IIG7
     .$I$+7IIIMMMG7I7                          ?G,DGGNII.
     .$~:$IIGMNGND77I:                          +I$GOD=?=
    ?=?IIIIGIIIINI7777?                        .7~=~===?.
  ONGNDG??IG?III$N7I777                          D~====I
.:$??7IIIIIII.....,....                          O::==~$D.
.,........ ..                                  ..M:I.==$7G.
                                                I?::IIIII7.
                                               .~:G:IIIIIG.
                                               .$:,O7III$$O
                                                 ::~DOGGNNO
                                                 .::,IOODI,
                                                   .7????$.
                                                     ... .
```

class `sage.misc.ascii_art.`**`AsciiArt`**(*lines=[ ]*, *breakpoints=[ ]*, *baseline=None*, *atomic=True*)

Bases: `sage.structure.sage_object.SageObject`

An Ascii art object is an object with some specific representation for *printing*.

INPUT:

- `lines` – the list of lines of the representation of the ascii art object
- `breakpoints` – the list of points where the representation can be split
- `baseline` – the reference line (from the bottom)
- `atomic` – indicate if the ascii art representation is splittable (must be coherent with breakpoints)

EXAMPLES:

```
sage: i = var('i')
sage: ascii_art(sum(pi^i/factorial(i)*x^i, i, 0, oo))
 pi*x
e
```

`get_baseline`()

Return the line where the baseline is, for example:

```
    5      4
14*x  + 5*x
```

the baseline has at line 0 and

```
{ o       }
{  \  : 4 }
{   o     }
```

has at line 1.

TESTS:

```
sage: from sage.misc.ascii_art import AsciiArt
sage: aa = AsciiArt(["   *   ", "  * *  ", " *   * ", "*******"], baseline=1);aa
   *
  * *
 *   *
*******
sage: aa.get_baseline()
1
sage: b = AsciiArt(["<-"])
sage: aa+b
   *
  * *
 *   * <-
*******
```

get_breakpoints()

Return an iterator of breakpoints where the object can be split.

For example the expression:

```
   5    4
14x + 5x
```

can be split on position 4 (on the +).

EXAMPLES:

```
sage: from sage.misc.ascii_art import AsciiArt
sage: p3 = AsciiArt([" * ", "***"])
sage: p5 = AsciiArt(["  *  ", " * * ", "*****"])
sage: aa = ascii_art([p3, p5])
sage: aa.get_breakpoints()
[2, 5, 6, 7, 12]
```

is_atomic()

Return `True` if the `AsciiArt` object is not splitable and `False` in otherwise.

For example, we considere a linear expression:

```
sage: a = 14*x^5 + 5*x^4
sage: ascii_art(a)
    5      4
14*x  + 5*x
```

If ASCII art object is not atomic, it is splittable on the + (in fact it is not really true because we use `sympy` to make ASCII art).

TESTS:

```
sage: from sage.misc.ascii_art import AsciiArt
sage: aa = AsciiArt(["  *  ", " * * ", "*****"]); aa
  *
 * *
*****
sage: aa.is_atomic()
True
sage: laa = ascii_art([aa,aa])
sage: laa.is_atomic()
False
```

split (*pos*)

Split the representation at the position `pos`.

EXMAPLES:

```
sage: from sage.misc.ascii_art import AsciiArt
sage: p3 = AsciiArt([" * ", "***"])
sage: p5 = AsciiArt(["  *  ", " * * ", "*****"])
sage: aa = ascii_art([p3, p5])
sage: a,b= aa.split(6)
sage: a
[
[  *
[ ***,
sage: b
   *   ]
  * *  ]
 ***** ]
```

sage.misc.ascii_art.**ascii_art** (*obj*)

Return an ASCII art reprensentation of `obj`:

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

TESTS:

```
sage: n = var('n')
sage: ascii_art(sum(binomial(2 * n, n + 1) * x^n, n, 0, oo))
 /        __________    \
-\2*x + \/ -4*x + 1  - 1/
--------------------------
           __________
     2*x*\/ -4*x + 1
sage: ascii_art(list(DyckWords(3)))
[                                   /\   ]
[            /\    /\      /\/\    /  \  ]
[ /\/\/\, /\/  \, /  \/\, /    \, /    \ ]
sage: ascii_art(1)
1
```

sage.misc.ascii_art.**ascii_art_dict** (*dict*)

Return an ASCII art output of a dictionnary.

TESTS:

```
sage: ascii_art({i:dw for i,dw in enumerate(DyckWords(3))})
{                                             /\   }
{                /\      /\        /\/\      /  \  }
{ 0:/\/\/\, 1:/\/  \, 2:/  \/\, 3:/    \, 4:/    \ }
```

`sage.misc.ascii_art.`**`ascii_art_list`**`(`*list*`)`

Return an ASCII art output of a list.

TESTS:

```
sage: ascii_art(list(DyckWords(3)))
[                                   /\   ]
[            /\    /\        /\/\    /  \  ]
[ /\/\/\, /\/  \, /  \/\, /    \, /    \ ]
```

`sage.misc.ascii_art.`**`ascii_art_set`**`(`*set*`)`

Return an ASCII art output of a set.

TESTS:

```
sage: ascii_art(set(DyckWords(3)))
{                                   /\   }
{  /\      /\/\            /\    /  \  }
{ /  \/\, /    \, /\/\/\, /\/  \, /    \ }
```

`sage.misc.ascii_art.`**`ascii_art_tuple`**`(`*tuple*`)`

Return an ASCII art output of a tuple.

TESTS:

```
sage: ascii_art(tuple(DyckWords(3)))
(                                   /\   )
(            /\    /\        /\/\    /  \  )
( /\/\/\, /\/  \, /  \/\, /    \, /    \ )
```

CHAPTER

THREE

BINDABLE CLASSES

class sage.misc.bindable_class.**BindableClass**

Bases: object

Bindable classes

This class implements a binding behavior for nested classes that derive from it. Namely, if a nested class Outer.Inner derives from BindableClass, and if outer is an instance of Outer, then outer.Inner(...) is equivalent to Outer.Inner(outer, ...).

EXAMPLES:

Let us consider the following class Outer with a nested class Inner:

```
sage: from sage.misc.nested_class import NestedClassMetaclass
sage: class Outer:
...       __metaclass__ = NestedClassMetaclass # just a workaround for Python misnaming nested c
...
...       class Inner:
...           def __init__(self, *args):
...               print args
...
...       def f(self, *args):
...           print self, args
...
...       @staticmethod
...       def f_static(*args):
...           print args
...
sage: outer = Outer()
```

By default, when Inner is a class nested in Outer, accessing outer.Inner returns the Inner class as is:

```
sage: outer.Inner is Outer.Inner
True
```

In particular, outer is completely ignored in the following call:

```
sage: x = outer.Inner(1,2,3)
(1, 2, 3)
```

This is similar to what happens with a static method:

```
sage: outer.f_static(1,2,3)
(1, 2, 3)
```

In some cases, we would want instead Inner` to receive outer as parameter, like in a usual method call:

```
sage: outer.f(1,2,3)
<__main__.Outer object at ...> (1, 2, 3)
```

To this end, `outer.f` returns a *bound method*:

```
sage: outer.f
<bound method Outer.f of <__main__.Outer object at ...>>
```

so that `outer.f(1,2,3)` is equivalent to:

```
sage: Outer.f(outer, 1,2,3)
<__main__.Outer object at ...> (1, 2, 3)
```

`BindableClass` gives this binding behavior to all its subclasses:

```
sage: from sage.misc.bindable_class import BindableClass
sage: class Outer:
...       __metaclass__ = NestedClassMetaclass # just a workaround for Python misnaming nested c
...
...       class Inner(BindableClass):
...           " some documentation "
...           def __init__(self, outer, *args):
...               print outer, args
```

Calling `Outer.Inner` returns the (unbound) class as usual:

```
sage: Outer.Inner
<class '__main__.Outer.Inner'>
```

However, `outer.Inner(1,2,3)` is equivalent to `Outer.Inner(outer, 1,2,3)`:

```
sage: outer = Outer()
sage: x = outer.Inner(1,2,3)
<__main__.Outer object at ...> (1, 2, 3)
```

To achieve this, `outer.Inner` returns (some sort of) bound class:

```
sage: outer.Inner
<bound class '__main__.Outer.Inner' of <__main__.Outer object at ...>>
```

Note: This is not actually a class, but an instance of `functools.partial`:

```
sage: type(outer.Inner).mro()
[<class 'sage.misc.bindable_class.BoundClass'>,
 <type 'functools.partial'>,
 <type 'object'>]
```

Still, documentation works as usual:

```
sage: outer.Inner.__doc__
' some documentation '
```

TESTS:

```
sage: from sage.misc.bindable_class import Outer
sage: TestSuite(Outer.Inner).run()
sage: outer = Outer()
sage: TestSuite(outer.Inner).run(skip=["_test_pickling"])
```

class `sage.misc.bindable_class.`**`BoundClass`**`(*args)`

Bases: `functools.partial`

class `sage.misc.bindable_class.`**`Inner2`**

Bases: `sage.misc.bindable_class.BindableClass`

Some documentation for Inner2

class `sage.misc.bindable_class.`**`Outer`**

Bases: `object`

A class with a bindable nested class, for testing purposes

class **`Inner`**

Bases: `sage.misc.bindable_class.BindableClass`

Some documentation for Outer.Inner

`Outer.`**`Inner2`**

alias of `Inner2`

CHAPTER

FOUR

CACHED FUNCTIONS AND METHODS

AUTHORS:

- William Stein: initial version, (inspired by conversation with Justin Walker)
- Mike Hansen: added doctests and made it work with class methods.
- Willem Jan Palenstijn: add CachedMethodCaller for binding cached methods to instances.
- Tom Boothby: added DiskCachedFunction.
- Simon King: improved performance, more doctests, cython version, CachedMethodCallerNoArgs, weak cached function, cached special methods.
- Julian Rueth (2014-03-19, 2014-05-09): added `key` parameter, allow caching for unhashable elements

EXAMPLES:

By trac ticket #11115, cached functions and methods are now also available in Cython code. The following examples cover various ways of usage.

Python functions:

```
sage: @cached_function
....: def test_pfunc(x):
....:     '''
....:     Some documentation
....:     '''
....:     return -x
sage: test_pfunc(5) is test_pfunc(5)
True
```

In some cases, one would only want to keep the result in cache as long as there is any other reference to the result. By trac ticket #12215, this is enabled for `UniqueRepresentation`, which is used to create unique parents: If an algebraic structure, such as a finite field, is only temporarily used, then it will not stay in cache forever. That behaviour is implemented using `weak_cached_function`, that behaves the same as `cached_function`, except that it uses a `WeakValueDictionary` for storing the results.

```
sage: from sage.misc.cachefunc import weak_cached_function
sage: class A: pass
sage: @weak_cached_function
....: def f():
....:     print "doing a computation"
....:     return A()
sage: a = f()
doing a computation
```

The result is cached:

```
sage: b = f()
sage: a is b
True
```

However, if there are no strong references left, the result may be garbage collected, and thus a new computation would take place:

```
sage: del a
sage: del b
sage: import gc
sage: n = gc.collect()
sage: a = f()
doing a computation
```

Cython cdef functions do not allow arbitrary decorators. However, one can wrap a Cython function and turn it into a cached function, by trac ticket #11115. We need to provide the name that the wrapped method or function should have, since otherwise the name of the original function would be used:

```
sage: cython('''cpdef test_funct(x): return -x''')
sage: wrapped_funct = cached_function(test_funct, name='wrapped_funct')
sage: wrapped_funct
Cached version of <built-in function test_funct>
sage: wrapped_funct.__name__
'wrapped_funct'
sage: wrapped_funct(5)
-5
sage: wrapped_funct(5) is wrapped_funct(5)
True
```

We can proceed similarly for cached methods of Cython classes, provided that they allow attribute assignment or have a public attribute `__cached_methods` of type `<dict>`. Since trac ticket #11115, this is the case for all classes inheriting from `Parent`. See below for a more explicit example. By trac ticket #12951, cached methods of extension classes can be defined by simply using the decorater. However, an indirect approach is still needed for cpdef methods:

```
sage: cython_code = ['cpdef test_meth(self,x):',
....: '    "some doc for a wrapped cython method"',
....: '    return -x',
....: 'from sage.all import cached_method',
....: 'from sage.structure.parent cimport Parent',
....: 'cdef class MyClass(Parent):',
....: '    @cached_method',
....: '    def direct_method(self, x):',
....: '        "Some doc for direct method"',
....: '        return 2*x',
....: '    wrapped_method = cached_method(test_meth,name="wrapped_method")']
sage: cython(os.linesep.join(cython_code))
sage: O = MyClass()
sage: O.direct_method
Cached version of <method 'direct_method' of '...MyClass' objects>
sage: O.wrapped_method
Cached version of <built-in function test_meth>
sage: O.wrapped_method.__name__
'wrapped_method'
sage: O.wrapped_method(5)
-5
sage: O.wrapped_method(5) is O.wrapped_method(5)
True
sage: O.direct_method(5)
```

```
10
sage: O.direct_method(5) is O.direct_method(5)
True
```

In some cases, one would only want to keep the result in cache as long as there is any other reference to the result. By trac ticket #12215, this is enabled for `UniqueRepresentation`, which is used to create unique parents: If an algebraic structure, such as a finite field, is only temporarily used, then it will not stay in cache forever. That behaviour is implemented using `weak_cached_function`, that behaves the same as `cached_function`, except that it uses a `WeakValueDictionary` for storing the results.

```
sage: from sage.misc.cachefunc import weak_cached_function
sage: class A: pass
sage: @weak_cached_function
....: def f():
....:     print "doing a computation"
....:     return A()
sage: a = f()
doing a computation
```

The result is cached:

```
sage: b = f()
sage: a is b
True
```

However, if there are no strong references left, the result may be garbage collected, and thus a new computation would take place:

```
sage: del a
sage: del b
sage: import gc
sage: n = gc.collect()
sage: a = f()
doing a computation
```

By trac ticket #11115, even if a parent does not allow attribute assignment, it can inherit a cached method from the parent class of a category (previously, the cache would have been broken):

```
sage: cython_code = ["from sage.all import cached_method, cached_in_parent_method, Category, Objects
....: "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()
```

In order to keep the memory footprint of elements small, it was decided to not support the same freedom of using cached methods for elements: If an instance of a class derived from `Element` does not allow attribute assignment, then a cached method inherited from the category of its parent will break, as in the class `MyBrokenElement` below.

However, there is a class `ElementWithCachedMethod` that has generally a slower attribute access, but fully supports cached methods. We remark, however, that cached methods are *much* faster if attribute access works. So, we expect that `ElementWithCachedMethod` will hardly by used.

```
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",
....: "        ElementWithCachedMethod.__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",
....: "from sage.structure.parent cimport Parent",
....: "cdef class MyParent(Parent):",
....: "    Element = MyElement"]
sage: cython('\n'.join(cython_code))
sage: P = MyParent(category=C)
sage: ebroken = MyBrokenElement(P,5)
sage: e = MyElement(P,5)
```

The cached methods inherited by the parent works:

```
sage: P.one()
<1>
sage: P.one() is P.one()
True
sage: P.invert(e)
<-5>
sage: P.invert(e) is P.invert(e)
True
```

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
```

The cache can be emptied:

```
sage: a = test_pfunc(5)
sage: test_pfunc.clear_cache()
sage: a is test_pfunc(5)
False
sage: a = P.one()
sage: P.one.clear_cache()
sage: a is P.one()
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
```

Introspection works:

```
sage: from sage.misc.edit_module import file_and_line
sage: from sage.misc.sageinspect import sage_getdoc, sage_getfile, sage_getsource
sage: print sage_getdoc(test_pfunc)
   Some documentation
sage: print sage_getdoc(O.wrapped_method)
some doc for a wrapped cython method

sage: print sage_getdoc(O.direct_method)
Some doc for direct method

sage: print sage_getsource(O.wrapped_method)
cpdef test_meth(self,x):
    "some doc for a wrapped cython method"
    return -x
sage: print sage_getsource(O.direct_method)
def direct_method(self, x):
    "Some doc for direct method"
    return 2*x
```

It is a very common special case to cache a method that has no arguments. In that special case, the time needed to access the cache can be drastically reduced by using a special implementation. The cached method decorator automatically determines which implementation ought to be chosen. A typical example is `sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal.gens()` (no arguments) versus `sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal.groebner_basis()` (several arguments):

```
sage: P.<a,b,c,d> = QQ[]
sage: I = P*[a,b]
sage: I.gens()
[a, b]
sage: I.gens() is I.gens()
True
sage: I.groebner_basis()
[a, b]
sage: I.groebner_basis() is I.groebner_basis()
True
sage: type(I.gens)
<type 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
sage: type(I.groebner_basis)
<type 'sage.misc.cachefunc.CachedMethodCaller'>
```

By trac ticket #12951, the cached_method decorator is also supported on non-c(p)def methods of extension classes, as long as they either support attribute assignment or have a public attribute of type `<dict>` called `__cached_methods`. The latter is easy:

```
sage: cython_code = [
....: "from sage.misc.cachefunc import cached_method",
....: "cdef class MyClass:",
....: "    cdef public dict __cached_methods",
....: "    @cached_method",
....: "    def f(self, a,b):",
....: "        return a*b"]
sage: cython(os.linesep.join(cython_code))
sage: P = MyClass()
sage: P.f(2,3)
6
sage: P.f(2,3) is P.f(2,3)
True
```

Providing attribute access is a bit more tricky, since it is needed that an attribute inherited by the instance from its class can be overridden on the instance. That is why providing a `__getattr__` would not be enough in the following example:

```
sage: cython_code = [
....: "from sage.misc.cachefunc import cached_method",
....: "cdef class MyOtherClass:",
....: "    cdef dict D",
....: "    def __init__(self):",
....: "        self.D = {}",
....: "    def __setattr__(self, n,v):",
....: "        self.D[n] = v",
....: "    def __getattribute__(self, n):",
....: "        try:",
....: "            return self.D[n]",
....: "        except KeyError:",
....: "            pass",
....: "        return getattr(type(self),n).__get__(self)",
....: "    @cached_method",
....: "    def f(self, a,b):",
....: "        return a+b"]
sage: cython(os.linesep.join(cython_code))
sage: Q = MyOtherClass()
sage: Q.f(2,3)
5
sage: Q.f(2,3) is Q.f(2,3)
True
```

Note that supporting attribute access is somehow faster than the easier method:

```
sage: timeit("a = P.f(2,3)")   # random
625 loops, best of 3: 1.3 µs per loop
sage: timeit("a = Q.f(2,3)")   # random
625 loops, best of 3: 931 ns per loop
```

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.add_bigoh(1)
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 method `_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)
```

Note: This attribute will only be accessed if the object itself is not hashable.

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.add_bigoh(1)
sage: b._cache_key() == d._cache_key() # this would be True if the parents were not included
False
```

class `sage.misc.cachefunc.`**`CachedFunction`**

Bases: `object`

Create a cached version of a function, which only recomputes values it hasn't already computed. Synonyme: `cached_function`

INPUT:

- `f` – a function
- `name` – (optional string) name that the cached version of `f` should be provided with
- `key` – (optional callable) takes the input and returns a key for the cache, typically one would use this to normalize input

If `f` is a function, do either `g = CachedFunction(f)` or `g = cached_function(f)` to make a cached version of `f`, or put `@cached_function` right before the definition of `f` (i.e., use Python decorators):

```
@cached_function
def f(...):
    ....
```

The inputs to the function must be hashable or they must define `sage.structure.sage_object.SageObject._cache_key()`.

EXAMPLES:

```
sage: @cached_function
....: def mul(x, y=2):
....:     return x*y
sage: mul(3)
6
```

We demonstrate that the result is cached, and that, moreover, the cache takes into account the various ways of providing default arguments:

```
sage: mul(3) is mul(3,2)
True
sage: mul(3,y=2) is mul(3,2)
True
```

The user can clear the cache:

```
sage: a = mul(4)
sage: mul.clear_cache()
sage: a is mul(4)
False
```

It is also possible to explicitly override the cache with a different value:

```
sage: mul.set_cache('foo',5)
sage: mul(5,2)
'foo'
```

The parameter `key` can be used to ignore parameters for caching. In this example we ignore the parameter `algorithm`:

```
sage: @cached_function(key=lambda x,y,algorithm: (x,y))
....: def mul(x, y, algorithm="default"):
....:     return x*y
sage: mul(1,1,algorithm="default") is mul(1,1,algorithm="algorithm") is mul(1,1) is mul(1,1,'def
True
```

cache

clear_cache()

Clear the cache dictionary.

EXAMPLES:

```
sage: g = CachedFunction(number_of_partitions)
sage: a = g(5)
sage: g.get_cache()
{((5, 'default'), ()): 7}
sage: g.clear_cache()
sage: g.get_cache()
{}
```

f

get_cache()

Returns the cache dictionary.

EXAMPLES:

```
sage: g = CachedFunction(number_of_partitions)
sage: a = g(5)
sage: g.get_cache()
{((5, 'default'), ()): 7}
```

get_key(**args*, ***kwds*)

Return the key in the cache to be used when `args` and `kwds` are passed in as parameters.

EXAMPLES:

```
sage: @cached_function
....: def foo(x):
....:     return x^2
sage: foo(2)
4
sage: foo.get_key(2)
((2,), ())
sage: foo.get_key(x=3)
((3,), ())
```

is_in_cache(**args*, ***kwds*)

Checks if the argument list is in the cache.

EXAMPLES:

```
sage: class Foo:
....:     def __init__(self, x):
....:         self._x = x
....:     @cached_method
....:     def f(self, z, y=0):
....:         return self._x*z+y
sage: a = Foo(2)
sage: a.f.is_in_cache(3)
False
sage: a.f(3)
6
sage: a.f.is_in_cache(3,y=0)
True
```

TESTS:

Check that trac ticket #16316 has been fixed, i.e., caching works for immutable unhashable objects which define `sage.structure.sage_object.SageObject._cache_key()`:

```
sage: @cached_function
....: def f(x): return x
sage: K.<u> = Qq(4)
sage: x = K(1,1); x
1 + O(2)
sage: f.is_in_cache(x)
False
sage: f(x)
1 + O(2)
sage: f.is_in_cache(x)
True
```

precompute(*arglist*, *num_processes=1*)

Cache values for a number of inputs. Do the computation in parallel, and only bother to compute values that we haven't already cached.

INPUT:

- `arglist` – list (or iterables) of arguments for which the method shall be precomputed.
- `num_processes` – number of processes used by `parallel()`

EXAMPLES:

```
sage: @cached_function
....: def oddprime_factors(n):
....:     l = [p for p,e in factor(n) if p != 2]
```

```
....:     return len(l)
sage: oddprime_factors.precompute(range(1,100), 4)
sage: oddprime_factors.cache[(25,),()]
1
```

set_cache (*value*, **args*, ***kwds*)

Set the value for those args and keyword args Mind the unintuitive syntax (value first). Any idea on how to improve that welcome!

EXAMPLES:

```
sage: g = CachedFunction(number_of_partitions)
sage: a = g(5)
sage: g.get_cache()
{((5, 'default'), ()): 7}
sage: g.set_cache(17, 5)
sage: g.get_cache()
{((5, 'default'), ()): 17}
sage: g(5)
17
```

TESTS:

Check that trac ticket #16316 has been fixed, i.e., caching works for immutable unhashable objects which define `sage.structure.sage_object.SageObject._cache_key()`:

```
sage: @cached_function
....: def f(x): return x
sage: K.<u> = Qq(4)
sage: x = K(1,1); x
1 + O(2)
sage: f.set_cache(x,x)
sage: f.is_in_cache(x)
True
```

DEVELOPER NOTE:

Is there a way to use the following intuitive syntax?

```
sage: g(5) = 19    # todo: not implemented
sage: g(5)         # todo: not implemented
19
```

class `sage.misc.cachefunc.`**`CachedInParentMethod`**

Bases: `sage.misc.cachefunc.CachedMethod`

A decorator that creates a cached version of an instance method of a class.

In contrast to `CachedMethod`, the cache dictionary is an attribute of the parent of the instance to which the method belongs.

ASSUMPTION:

This way of caching works only if

- the instances *have* a parent, and
- the instances are hashable (they are part of the cache key) or they define `sage.structure.sage_object.SageObject._cache_key()`

NOTE:

For proper behavior, the method must be a pure function (no side effects). If this decorator is used on a method, it will have identical output on equal elements. This is since the element is part of the hash key. Arguments to the method must be hashable or define `sage.structure.sage_object.SageObject._cache_key()`. The instance it is assigned to must be hashable.

Examples can be found at `cachefunc`.

class `sage.misc.cachefunc.`**`CachedMethod`**

Bases: `object`

A decorator that creates a cached version of an instance method of a class.

Note: For proper behavior, the method must be a pure function (no side effects). Arguments to the method must be hashable or transformed into something hashable using `key` or they must define `sage.structure.sage_object.SageObject._cache_key()`.

EXAMPLES:

```
sage: class Foo(object):
....:     @cached_method
....:     def f(self, t, x=2):
....:         print 'computing'
....:         return t**x
sage: a = Foo()
```

The example shows that the actual computation takes place only once, and that the result is identical for equivalent input:

```
sage: res = a.f(3, 2); res
computing
9
sage: a.f(t = 3, x = 2) is res
True
sage: a.f(3) is res
True
```

Note, however, that the `CachedMethod` is replaced by a `CachedMethodCaller` or `CachedMethodCallerNoArgs` as soon as it is bound to an instance or class:

```
sage: P.<a,b,c,d> = QQ[]
sage: I = P*[a,b]
sage: type(I.__class__.gens)
<type 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
```

So, you would hardly ever see an instance of this class alive.

The parameter `key` can be used to pass a function which creates a custom cache key for inputs. In the following example, this parameter is used to ignore the `algorithm` keyword for caching:

```
sage: class A(object):
....:     def _f_normalize(self, x, algorithm): return x
....:     @cached_method(key=_f_normalize)
....:     def f(self, x, algorithm='default'): return x
sage: a = A()
sage: a.f(1, algorithm="default") is a.f(1) is a.f(1, algorithm="algorithm")
True
```

class `sage.misc.cachefunc.`**`CachedMethodCaller`**

Bases: `sage.misc.cachefunc.CachedFunction`

Utility class that is used by `CachedMethod` to bind a cached method to an instance.

Note: Since trac ticket #11115, there is a special implementation `CachedMethodCallerNoArgs` for methods that do not take arguments.

EXAMPLE:

```
sage: class A:
....:     @cached_method
....:     def bar(self,x):
....:         return x^2
sage: a = A()
sage: a.bar
Cached version of <function bar at 0x...>
sage: type(a.bar)
<type 'sage.misc.cachefunc.CachedMethodCaller'>
sage: a.bar(2) is a.bar(x=2)
True
```

get_key (**args*, ***kwds*)

Convert arguments to the key for this instance's cache.

EXAMPLES:

```
sage: class Foo:
....:     def __init__(self, x):
....:         self._x = x
....:     @cached_method
....:     def f(self, y, z=0):
....:         return self._x * y + z
sage: a = Foo(2)
sage: z = a.f(37)
sage: k = a.f.get_key(37); k
((37, 0), ())
sage: a.f.get_cache()[k] is z
True
```

Note that the method does not test whether there are too many arguments, or wrong argument names:

```
sage: a.f.get_key(1,2,3,x=4,y=5,z=6)
((1, 2, 3), (('x', 4), ('y', 5), ('z', 6)))
```

It does, however, take into account the different ways of providing named arguments, possibly with a default value:

```
sage: a.f.get_key(5)
((5, 0), ())
sage: a.f.get_key(y=5)
((5, 0), ())
sage: a.f.get_key(5,0)
((5, 0), ())
sage: a.f.get_key(5,z=0)
((5, 0), ())
sage: a.f.get_key(y=5,z=0)
((5, 0), ())
```

precompute (*arglist*, *num_processes=1*)

Cache values for a number of inputs. Do the computation in parallel, and only bother to compute values that we haven't already cached.

INPUT:

- `arglist` – list (or iterables) of arguments for which the method shall be precomputed.
- `num_processes` – number of processes used by `parallel()`

EXAMPLES:

```
sage: class Foo(object):
....:     @cached_method
....:     def f(self, i):
....:         return i^2
sage: foo = Foo()
sage: foo.f(3)
9
sage: foo.f(1)
1
sage: foo.f.precompute(range(2), 2)
sage: foo.f.cache
{((0,), ()): 0, ((1,), ()): 1, ((3,), ()): 9}
```

class `sage.misc.cachefunc.`**`CachedMethodCallerNoArgs`**

Bases: `sage.misc.cachefunc.CachedFunction`

Utility class that is used by `CachedMethod` to bind a cached method to an instance, in the case of a method that does not accept any arguments except `self`.

Note: The return value `None` would not be cached. So, if you have a method that does not accept arguments and may return `None` after a lengthy computation, then `@cached_method` should not be used.

EXAMPLE:

```
sage: P.<a,b,c,d> = QQ[]
sage: I = P*[a,b]
sage: I.gens
Cached version of <function gens at 0x...>
sage: type(I.gens)
<type 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
sage: I.gens is I.gens
True
sage: I.gens() is I.gens()
True
```

AUTHOR:

- Simon King (2011-04)

`clear_cache`()

Clear the cache dictionary.

EXAMPLES:

```
sage: P.<a,b,c,d> = QQ[]
sage: I = P*[a,b]
sage: I.gens()
[a, b]
sage: I.gens.set_cache('bar')
sage: I.gens()
'bar'
```

The cache can be emptied and thus the original value will be reconstructed:

```
sage: I.gens.clear_cache()
sage: I.gens()
[a, b]
```

is_in_cache()

Answers whether the return value is already in the cache.

Note: Recall that a cached method without arguments can not cache the return value `None`.

EXAMPLE:

```
sage: P.<x,y> = QQ[]
sage: I = P*[x,y]
sage: I.gens.is_in_cache()
False
sage: I.gens()
[x, y]
sage: I.gens.is_in_cache()
True
```

set_cache(*value*)

Override the cache with a specific value.

Note: `None` is not suitable for a cached value. It would be interpreted as an empty cache, forcing a new computation.

EXAMPLES:

```
sage: P.<a,b,c,d> = QQ[]
sage: I = P*[a,b]
sage: I.gens()
[a, b]
sage: I.gens.set_cache('bar')
sage: I.gens()
'bar'
```

The cache can be emptied and thus the original value will be reconstructed:

```
sage: I.gens.clear_cache()
sage: I.gens()
[a, b]
```

The attempt to assign `None` to the cache fails:

```
sage: I.gens.set_cache(None)
sage: I.gens()
[a, b]
```

class sage.misc.cachefunc.**CachedMethodPickle**(*inst*, *name*, *cache=None*)

Bases: `object`

This class helps to unpickle cached methods.

Note: Since trac ticket #8611, a cached method is an attribute of the instance (provided that it has a `__dict__`). Hence, when pickling the instance, it would be attempted to pickle that attribute as well, but this is a problem, since functions can not be pickled, currently. Therefore, we replace the actual cached method by a place holder, that kills itself as soon as any attribute is requested. Then, the original cached attribute is reinstated. But the cached values are in fact saved.

EXAMPLES:

```
sage: R.<x, y, z> = PolynomialRing(QQ, 3)
sage: I = R*(x^3 + y^3 + z^3,x^4-y^4)
sage: I.groebner_basis()
[y^5*z^3 - 1/4*x^2*z^6 + 1/2*x*y*z^6 + 1/4*y^2*z^6,
 x^2*y*z^3 - x*y^2*z^3 + 2*y^3*z^3 + z^6,
 x*y^3 + y^4 + x*z^3, x^3 + y^3 + z^3]
sage: I.groebner_basis
Cached version of <function groebner_basis at 0x...>
```

We now pickle and unpickle the ideal. The cached method `groebner_basis` is replaced by a placeholder:

```
sage: J = loads(dumps(I))
sage: J.groebner_basis
Pickle of the cached method "groebner_basis"
```

But as soon as any other attribute is requested from the placeholder, it replaces itself by the cached method, and the entries of the cache are actually preserved:

```
sage: J.groebner_basis.is_in_cache()
True
sage: J.groebner_basis
Cached version of <function groebner_basis at 0x...>
sage: J.groebner_basis() == I.groebner_basis()
True
```

TESTS:

Since trac ticket #11115, there is a special implementation for cached methods that don't take arguments:

```
sage: P.<a,b,c,d> = QQ[]
sage: I = P*[a,b]
sage: type(I.gens)
<type 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
sage: type(I.groebner_basis)
<type 'sage.misc.cachefunc.CachedMethodCaller'>
```

We demonstrate that both implementations can be pickled and preserve the cache. For that purpose, we assign nonsense to the cache. Of course, it is a very bad idea to override the cache in that way. So, please don't try this at home:

```
sage: I.groebner_basis.set_cache('foo',algorithm='singular')
sage: I.groebner_basis(algorithm='singular')
'foo'
sage: I.gens.set_cache('bar')
sage: I.gens()
'bar'
sage: J = loads(dumps(I))
sage: J.gens()
'bar'
sage: J.groebner_basis(algorithm='singular')
'foo'
```

Anyway, the cache will be automatically reconstructed after clearing it:

```
sage: J.gens.clear_cache()
sage: J.gens()
[a, b]
sage: J.groebner_basis.clear_cache()
```

```
sage: J.groebner_basis(algorithm='singular')
[a, b]
```

AUTHOR:

- Simon King (2011-01)

class sage.misc.cachefunc.**CachedSpecialMethod**

Bases: sage.misc.cachefunc.CachedMethod

Cached version of *special* python methods.

IMPLEMENTATION:

For new style classes C, it is not possible to override a special method, such as __hash__, in the __dict__ of an instance c of C, because Python will for efficiency reasons always use what is provided by the class, not by the instance.

By consequence, if __hash__ would be wrapped by using CachedMethod, then hash(c) will access C.__hash__ and bind it to c, which means that the __get__ method of CachedMethod will be called. But there, we assume that Python has already inspected __dict__, and thus a CachedMethodCaller will be created over and over again.

Here, the __get__ method will explicitly access the __dict__, so that hash(c) will rely on a single CachedMethodCaller stored in the __dict__.

EXAMPLES:

```
sage: class C:
....:     @cached_method
....:     def __hash__(self):
....:         print "compute hash"
....:         return int(5)
....:
sage: c = C()
sage: type(C.__hash__)
<type 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
```

The hash is computed only once, subsequent calls will use the value from the cache. This was implemented in trac ticket #12601.

```
sage: hash(c)       # indirect doctest
compute hash
5
sage: hash(c)
5
```

class sage.misc.cachefunc.**ClearCacheOnPickle**

Bases: object

This class implements an appropriate __getstate__ method that clears the cache of the methods (see @cached_method) before passing them on to the caller, typically the pickle and copy modules.

The implemented __getstate__ method calls the __getstate__ methods of classes later in the method resolution order. Therefore, classes which want this behaviour should inherit first from this one.

EXAMPLE:

In the following example, we create a Python class that inherits from multivariate polynomial ideals, but does not pickle cached values. We provide the definition in Cython, however, since interactive Cython definitions provide introspection by trac ticket #9976, whereas Python definitions don't.

```
sage: P.<a,b,c,d> = QQ[]
sage: I = P*[a,b]
sage: classdef = ['from sage.misc.cachefunc import ClearCacheOnPickle',
....:    'from sage.all import QQ',
....:    'P = QQ["a","b","c","d"]; I = P*[P.gen(0),P.gen(1)]',
....:    'class MyClass(ClearCacheOnPickle,I.__class__):',
....:    '    def __init__(self,ring,gens):',
....:    '        I.__class__.__init__(self,ring,gens)',
....:    '    def __getnewargs__(self):',
....:    '        return (self._Ideal_generic__ring,self._Ideal_generic__gens)']
sage: cython('\n'.join(classdef))
```

We destroy the cache of two methods of `I` on purpose (demonstrating that the two different implementations of cached methods are correctly dealt with). Pickling `I` preserves the cache:

```
sage: I.gens.set_cache('bar')
sage: I.groebner_basis.set_cache('foo',algorithm='singular')
sage: J = loads(dumps(I))
sage: J.gens()
'bar'
sage: J.groebner_basis(algorithm='singular')
'foo'
```

However, if we have an ideal that additionally descends from `ClearCacheOnPickle`, the carefully corrupted cache is not pickled:

```
sage: A = MyClass(P,[a,b])
sage: A
Ideal (a, b) of Multivariate Polynomial Ring in a, b, c, d over Rational Field
sage: A.gens.set_cache('foo')
sage: A.groebner_basis.set_cache('bar',algorithm='singular')
sage: A.gens()
'foo'
sage: A.groebner_basis(algorithm='singular')
'bar'
sage: B = loads(dumps(A))
sage: B.gens()
[a, b]
sage: B.groebner_basis(algorithm='singular')
[a, b]
sage: A.gens()
'foo'
```

class `sage.misc.cachefunc.`**`DiskCachedFunction`**(*f*, *dir*, *memory_cache=False*, *key=None*)

Bases: `sage.misc.cachefunc.CachedFunction`

Works similar to CachedFunction, but instead, we keep the cache on disk (optionally, we keep it in memory too).

EXAMPLES:

```
sage: from sage.misc.cachefunc import DiskCachedFunction
sage: dir = tmp_dir()
sage: factor = DiskCachedFunction(factor, dir, memory_cache=True)
sage: f = factor(2775); f
3 * 5^2 * 37
sage: f is factor(2775)
True
```

class sage.misc.cachefunc.**FileCache**(*dir*, *prefix='‘*, *memory_cache=False*)

FileCache is a dictionary-like class which stores keys and values on disk. The keys take the form of a tuple (A,K)

- A is a tuple of objects t where each t is an exact object which is uniquely identified by a short string.
- K is a tuple of tuples (s,v) where s is a valid variable name and v is an exact object which is uniquely identified by a short string with letters [a-zA-Z0-9-._]

The primary use case is the DiskCachedFunction. If memory_cache == True, we maintain a cache of objects seen during this session in memory – but we don't load them from disk until necessary. The keys and values are stored in a pair of files:

- prefix-argstring.key.sobj contains the key only,
- prefix-argstring.sobj contains the tuple (key,val)

where self[key] == val.

Note: We assume that each FileCache lives in its own directory. Use **extreme** caution if you wish to break that assumption.

---

file_list()

Return the list of files corresponding to self.

EXAMPLES:

```
sage: from sage.misc.cachefunc import FileCache
sage: dir = tmp_dir()
sage: FC = FileCache(dir, memory_cache = True, prefix='t')
sage: FC[((),())] = 1
sage: FC[((1,2),())] = 2
sage: FC[((1,),(('a',1),))] = 3
sage: for f in sorted(FC.file_list()): print f[len(dir):]
t-.key.sobj
t-.sobj
t-1_2.key.sobj
t-1_2.sobj
t-a-1.1.key.sobj
t-a-1.1.sobj
```

items()

Return a list of tuples (k,v) where self[k] = v.

EXAMPLES:

```
sage: from sage.misc.cachefunc import FileCache
sage: dir = tmp_dir()
sage: FC = FileCache(dir, memory_cache = False)
sage: FC[((),())] = 1
sage: FC[((1,2),())] = 2
sage: FC[((1,),(('a',1),))] = 3
sage: I = FC.items()
sage: I.sort(); print I
[(((), ()), 1), (((1,), (('a', 1),)), 3), (((1, 2), ()), 2)]
```

keys()

Return a list of keys k where self[k] is defined.

EXAMPLES:

```
sage: from sage.misc.cachefunc import FileCache
sage: dir = tmp_dir()
sage: FC = FileCache(dir, memory_cache = False)
sage: FC[((),())] = 1
sage: FC[((1,2),())] = 2
sage: FC[((1,),(('a',1),))] = 3
sage: K = FC.keys()
sage: K.sort(); print K
[((), ()), ((1,), (('a', 1),)), ((1, 2), ())]
```

values()

Return a list of values that are stored in `self`.

EXAMPLES:

```
sage: from sage.misc.cachefunc import FileCache
sage: dir = tmp_dir()
sage: FC = FileCache(dir, memory_cache = False)
sage: FC[((),())] = 1
sage: FC[((1,2),())] = 2
sage: FC[((1,),(('a',1),))] = 3
sage: FC[((),(('a',1),))] = 4
sage: v = FC.values()
sage: v.sort(); print v
[1, 2, 3, 4]
```

class `sage.misc.cachefunc.`**WeakCachedFunction**

Bases: `sage.misc.cachefunc.CachedFunction`

A version of `CachedFunction` using weak references on the values.

If `f` is a function, do either `g = weak_cached_function(f)` to make a cached version of `f`, or put `@weak_cached_function` right before the definition of `f` (i.e., use Python decorators):

```
@weak_cached_function
def f(...):
    ...
```

EXAMPLES:

```
sage: from sage.misc.cachefunc import weak_cached_function
sage: class A: pass
sage: @weak_cached_function
....: def f():
....:     print "doing a computation"
....:     return A()
sage: a = f()
doing a computation
```

The result is cached:

```
sage: b = f()
sage: a is b
True
```

However, if there are no strong references left, the result may be garbage collected, and thus a new computation would take place:

```
sage: del a
sage: del b
sage: import gc
```

```
sage: n = gc.collect()
sage: a = f()
doing a computation
```

The parameter `key` can be used to ignore parameters for caching. In this example we ignore the parameter `algorithm`:

```
sage: @weak_cached_function(key=lambda x,algorithm: x)
....: def mod_ring(x, algorithm="default"):
....:     return IntegerModRing(x)
sage: mod_ring(1,algorithm="default") is mod_ring(1,algorithm="algorithm") is mod_ring(1) is mod
True
```

is_in_cache(**args*, ***kwds*)

Check if the argument list is in the cache.

EXAMPLES:

```
sage: from sage.misc.cachefunc import weak_cached_function
sage: class A:
....:     def __init__(self, x):
....:         self.x = x
sage: @weak_cached_function
....: def f(n):
....:    return A(n)
sage: a = f(5)
```

The key 5 is in the cache, as long as there is a strong reference to the corresponding value:

```
sage: f.is_in_cache(5)
True
```

However, if there are no strong references left, the cached item is removed from cache after garbage collection:

```
sage: del a
sage: import gc
sage: n = gc.collect()
sage: f.is_in_cache(5)
False
```

TESTS:

Check that trac ticket #16316 has been fixed, i.e., caching works for immutable unhashable objects which define `sage.structure.sage_object.SageObject._cache_key()`:

```
sage: from sage.misc.cachefunc import weak_cached_function
sage: @weak_cached_function
....: def f(x): return x
sage: K.<u> = Qq(4)
sage: R.<t> = K[]
sage: f.is_in_cache(t)
False
sage: f(t)
(1 + O(2^20))*t
sage: f.is_in_cache(t)
True
```

set_cache(*value*, **args*, ***kwds*)

Set the value for those args and keyword args Mind the unintuitive syntax (value first). Any idea on how to improve that welcome!

It is required that the given value is weak referenceable. The item will be removed from cache if the value is garbage collected.

EXAMPLES:

```
sage: from sage.misc.cachefunc import weak_cached_function
sage: @weak_cached_function
....: def f(n):
....:     raise RuntimeError
sage: f.set_cache(ZZ, 5)
sage: f(5)
Integer Ring
```

TESTS:

Check that trac ticket #16316 has been fixed, i.e., caching works for immutable unhashable objects which define `sage.structure.sage_object.SageObject._cache_key()`:

```
sage: from sage.misc.cachefunc import weak_cached_function
sage: @weak_cached_function
....: def f(x): return x
sage: K.<u> = Qq(4)
sage: R.<t> = K[]
sage: f.set_cache(t,t)
sage: f.is_in_cache(t)
True
```

`sage.misc.cachefunc.`**`cached_function`**(*self*, **args*, ***kwds*)

Create a cached version of a function, which only recomputes values it hasn't already computed. Synonyme: `cached_function`

INPUT:

- `f` – a function
- `name` – (optional string) name that the cached version of `f` should be provided with
- `key` – (optional callable) takes the input and returns a key for the cache, typically one would use this to normalize input

If `f` is a function, do either `g = CachedFunction(f)` or `g = cached_function(f)` to make a cached version of `f`, or put `@cached_function` right before the definition of `f` (i.e., use Python decorators):

```
@cached_function
def f(...):
    ....
```

The inputs to the function must be hashable or they must define `sage.structure.sage_object.SageObject._cache_key()`.

EXAMPLES:

```
sage: @cached_function
....: def mul(x, y=2):
....:     return x*y
sage: mul(3)
6
```

We demonstrate that the result is cached, and that, moreover, the cache takes into account the various ways of providing default arguments:

```
sage: mul(3) is mul(3,2)
True
sage: mul(3,y=2) is mul(3,2)
True
```

The user can clear the cache:

```
sage: a = mul(4)
sage: mul.clear_cache()
sage: a is mul(4)
False
```

It is also possible to explicitly override the cache with a different value:

```
sage: mul.set_cache('foo',5)
sage: mul(5,2)
'foo'
```

The parameter `key` can be used to ignore parameters for caching. In this example we ignore the parameter `algorithm`:

```
sage: @cached_function(key=lambda x,y,algorithm: (x,y))
....: def mul(x, y, algorithm="default"):
....:     return x*y
sage: mul(1,1,algorithm="default") is mul(1,1,algorithm="algorithm") is mul(1,1) is mul(1,1,'def
True
```

`sage.misc.cachefunc.`**`cached_in_parent_method`**(*self*, *inst*, *args, **kwds)

A decorator that creates a cached version of an instance method of a class.

In contrast to `CachedMethod`, the cache dictionary is an attribute of the parent of the instance to which the method belongs.

ASSUMPTION:

This way of caching works only if

- the instances *have* a parent, and
- the instances are hashable (they are part of the cache key) or they define `sage.structure.sage_object.SageObject._cache_key()`

NOTE:

For proper behavior, the method must be a pure function (no side effects). If this decorator is used on a method, it will have identical output on equal elements. This is since the element is part of the hash key. Arguments to the method must be hashable or define `sage.structure.sage_object.SageObject._cache_key()`. The instance it is assigned to must be hashable.

Examples can be found at `cachefunc`.

`sage.misc.cachefunc.`**`cached_method`**(*f*, *name=None*, *key=None*)

A decorator for cached methods.

EXAMPLES:

In the following examples, one can see how a cached method works in application. Below, we demonstrate what is done behind the scenes:

```
sage: class C:
....:     @cached_method
....:     def __hash__(self):
....:         print "compute hash"
```

```
....:         return int(5)
....:     @cached_method
....:     def f(self, x):
....:         print "computing cached method"
....:         return x*2
sage: c = C()
sage: type(C.__hash__)
<type 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
sage: hash(c)
compute hash
5
```

When calling a cached method for the second time with the same arguments, the value is gotten from the cache, so that a new computation is not needed:

```
sage: hash(c)
5
sage: c.f(4)
computing cached method
8
sage: c.f(4) is c.f(4)
True
```

Different instances have distinct caches:

```
sage: d = C()
sage: d.f(4) is c.f(4)
computing cached method
False
sage: d.f.clear_cache()
sage: c.f(4)
8
sage: d.f(4)
computing cached method
8
```

Using cached methods for the hash and other special methods was implemented in trac ticket #12601, by means of `CachedSpecialMethod`. We show that it is used behind the scenes:

```
sage: cached_method(c.__hash__)
<sage.misc.cachefunc.CachedSpecialMethod object at ...>
sage: cached_method(c.f)
<sage.misc.cachefunc.CachedMethod object at ...>
```

class `sage.misc.cachefunc.`**`disk_cached_function`**(*dir*, *memory_cache=False*, *key=None*)

Decorator for `DiskCachedFunction`.

EXAMPLES:

```
sage: dir = tmp_dir()
sage: @disk_cached_function(dir)
....: def foo(x): return next_prime(2^x)%x
sage: x = foo(200);x
11
sage: @disk_cached_function(dir)
....: def foo(x): return 1/x
sage: foo(200)
11
sage: foo.clear_cache()
sage: foo(200)
```

```
1/200
```

sage.misc.cachefunc.**weak_cached_function**(*self*, **args*, ***kwds*)

A version of `CachedFunction` using weak references on the values.

If `f` is a function, do either `g = weak_cached_function(f)` to make a cached version of `f`, or put `@weak_cached_function` right before the definition of `f` (i.e., use Python decorators):

```
@weak_cached_function
def f(...):
    ...
```

EXAMPLES:

```
sage: from sage.misc.cachefunc import weak_cached_function
sage: class A: pass
sage: @weak_cached_function
....: def f():
....:     print "doing a computation"
....:     return A()
sage: a = f()
doing a computation
```

The result is cached:

```
sage: b = f()
sage: a is b
True
```

However, if there are no strong references left, the result may be garbage collected, and thus a new computation would take place:

```
sage: del a
sage: del b
sage: import gc
sage: n = gc.collect()
sage: a = f()
doing a computation
```

The parameter `key` can be used to ignore parameters for caching. In this example we ignore the parameter `algorithm`:

```
sage: @weak_cached_function(key=lambda x,algorithm: x)
....: def mod_ring(x, algorithm="default"):
....:     return IntegerModRing(x)
sage: mod_ring(1,algorithm="default") is mod_ring(1,algorithm="algorithm") is mod_ring(1) is mod
True
```

CHAPTER

FIVE

FAST AND SAFE WEAK VALUE DICTIONARY

AUTHORS:

- Simon King (2013-10)
- Nils Bruin (2013-10)
- Julian Rueth (2014-03-16): improved handling of unhashable objects

Python's `weakref` module provides `WeakValueDictionary`. This behaves similar to a dictionary, but it does not prevent its values from garbage collection. Hence, it stores the values by weak references with callback functions: The callback function deletes a key-value pair from the dictionary, as soon as the value becomes subject to garbage collection.

However, a problem arises if hash and comparison of the key depend on the value that is being garbage collected:

```
sage: import weakref
sage: class Vals(object): pass
sage: class Keys:
....:     def __init__(self, val):
....:         self.val = weakref.ref(val)
....:     def __hash__(self):
....:         return hash(self.val())
....:     def __eq__(self, other):
....:         return self.val() == other.val()
sage: ValList = [Vals() for _ in range(10)]
sage: D = weakref.WeakValueDictionary()
sage: for v in ValList:
....:     D[Keys(v)] = v
sage: len(D)
10
sage: del ValList, v
Exception KeyError: (<__main__.Keys instance at ...>,) in <function remove at ...> ignored
Exception KeyError: (<__main__.Keys instance at ...>,) in <function remove at ...> ignored
Exception KeyError: (<__main__.Keys instance at ...>,) in <function remove at ...> ignored
Exception KeyError: (<__main__.Keys instance at ...>,) in <function remove at ...> ignored
...
sage: len(D) > 1
True
```

Hence, there are scary error messages, and moreover the defunct items have not been removed from the dictionary.

Therefore, Sage provides an alternative implementation `sage.misc.weak_dict.WeakValueDictionary`, using a callback that removes the defunct item not based on hash and equality check of the key (this is what fails in the example above), but based on comparison by identity. This is possible, since references with callback function are distinct even if they point to the same object. Hence, even if the same object `O` occurs as value for several keys, each reference to `O` corresponds to a unique key. We see no error messages, and the items get correctly removed:

```
sage: ValList = [Vals() for _ in range(10)]
sage: import sage.misc.weak_dict
sage: D = sage.misc.weak_dict.WeakValueDictionary()
sage: for v in ValList:
....:     D[Keys(v)] = v
sage: len(D)
10
sage: del ValList
sage: len(D)
1
sage: del v
sage: len(D)
0
```

Another problem arises when iterating over the items of a dictionary: If garbage collection occurs during iteration, then the content of the dictionary changes, and the iteration breaks for `weakref.WeakValueDictionary`:

```
sage: class Cycle:
....:     def __init__(self):
....:         self.selfref = self
sage: C = [Cycle() for n in range(10)]
sage: D = weakref.WeakValueDictionary(enumerate(C))
sage: import gc
sage: gc.disable()
sage: del C[:5]
sage: len(D)
10
```

With `WeakValueDictionary`, the behaviour is safer. Note that iteration over a WeakValueDictionary is non-deterministic, since the lifetime of values (and hence the presence of keys) in the dictionary may depend on when garbage collection occurs. The method implemented here will at least postpone dictionary mutations due to garbage collection callbacks. This means that as long as there is at least one iterator active on a dictionary, none of its keys will be deallocated (which could have side-effects). Which entries are returned is of course still dependent on when garbage collection occurs. Note that when a key gets returned as "present" in the dictionary, there is no guarantee one can actually retrieve its value: it may have been garbage collected in the mean time.

Note that Sage's weak value dictionary is actually an instance of `dict`, in contrast to `weakref`'s weak value dictionary:

```
sage: issubclass(weakref.WeakValueDictionary, dict)
False
sage: issubclass(sage.misc.weak_dict.WeakValueDictionary, dict)
True
```

See trac ticket #13394 for a discussion of some of the design considerations.

class `sage.misc.weak_dict.`**`WeakValueDictEraser`**

Bases: `object`

Erases items from a `sage.misc.weak_dict.WeakValueDictionary` 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.misc.weak_dict import WeakValueDictionary
sage: v = frozenset([1])
sage: D = WeakValueDictionary({1 : v})
```

```
sage: len(D)
1
sage: del v
sage: len(D)
0
```

AUTHOR:

- Nils Bruin (2013-11)

class sage.misc.weak_dict.**WeakValueDictionary**

Bases: dict

IMPLEMENTATION:

The WeakValueDictionary inherits from dict. In its implementation, it stores weakrefs to the actual values under the keys. All access routines are wrapped to transparently place and remove these weakrefs.

NOTE:

In contrast to weakref.WeakValueDictionary in Python's weakref module, the callback does not need to assume that the dictionary key is a valid Python object when it is called. There is no need to compute the hash or compare the dictionary keys. This is why the example below would not work with weakref.WeakValueDictionary, but does work with sage.misc.weak_dict.WeakValueDictionary.

EXAMPLES:

```
sage: import weakref
sage: class Vals(object): pass
sage: class Keys:
....:     def __init__(self, val):
....:         self.val = weakref.ref(val)
....:     def __hash__(self):
....:         return hash(self.val())
....:     def __eq__(self, other):
....:         return self.val() == other.val()
sage: ValList = [Vals() for _ in range(10)]
sage: import sage.misc.weak_dict
sage: D = sage.misc.weak_dict.WeakValueDictionary()
sage: for v in ValList:
....:     D[Keys(v)] = v
sage: len(D)
10
sage: del ValList
sage: len(D)
1
sage: del v
sage: len(D)
0
```

TESTS:

The following reflects the behaviour of the callback on weak dict values, as discussed on trac ticket #13394.

```
sage: from sage.misc.weak_dict import WeakValueDictionary
sage: V = [set(range(n)) for n in range(5)]
sage: D = WeakValueDictionary(enumerate(V))
```

The line V[k] = None triggers execution of the callback functions of the dictionary values. However, the actual deletion is postponed till after the iteration over the dictionary has finished. Hence, when the callbacks are executed, the values which the callback belongs to has already been overridded by a new value. Therefore, the callback does not delete the item:

```
sage: for k in D.iterkeys():    # indirect doctest
....:     V[k] = None
....:     D[k] = ZZ
sage: len(D)
5
sage: D[1]
Integer Ring
```

The following is a stress test for weak value dictionaries:

```
sage: class C(object):
....:     def __init__(self, n):
....:         self.n = n
....:     def __cmp__(self, other):
....:         return cmp(type(self),type(other)) or cmp(self.n, other.n)
sage: B = 100
sage: L = [None]*B
sage: D1 = WeakValueDictionary()
sage: D2 = WeakValueDictionary()
sage: for i in range(10000):
....:     ki = floor(random()*B)
....:     vi = C(floor(random()*B))
....:     D1[ki] = vi
....:     D2[ki] = vi
....:     L[ki]  = vi
....:     del vi
....:     ko = floor(random()*B)
....:     if ko in D1:
....:         del D1[ko]
....:         L[ko] = None
....:     assert D1 == D2
```

get (*k*, *d=None*)

Return the stored value for a key, or a default value for unkown keys.

The default value defaults to None.

EXAMPLES:

```
sage: import sage.misc.weak_dict
sage: L = [GF(p) for p in prime_range(10^3)]
sage: D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))
sage: 100 in D
True
sage: 200 in D
False
sage: D.get(100, "not found")
Finite Field of size 547
sage: D.get(200, "not found")
'not found'
sage: D.get(200) is None
True
```

TESTS:

Check that trac ticket #15956 has been fixed, i.e., a `TypeError` is raised for unhashable objects:

```
sage: D = sage.misc.weak_dict.WeakValueDictionary()
sage: D.get(matrix([]))
Traceback (most recent call last):
...
TypeError: mutable matrices are unhashable
```

items()

The key-value pairs of this dictionary.

EXAMPLES:

```
sage: import sage.misc.weak_dict
sage: class Vals:
....:     def __init__(self, n):
....:         self.n = n
....:     def __repr__(self):
....:         return "<%s>"%self.n
....:     def __cmp__(self, other):
....:         c = cmp(type(self),type(other))
....:         if c:
....:             return c
....:         return cmp(self.n,other.n)
sage: class Keys(object):
....:     def __init__(self, n):
....:         self.n = n
....:     def __hash__(self):
....:         if self.n%2:
....:             return 5
....:         return 3
....:     def __repr__(self):
....:         return "[%s]"%self.n
....:     def __cmp__(self, other):
....:         c = cmp(type(self),type(other))
....:         if c:
....:             return c
....:         return cmp(self.n,other.n)
sage: L = [(Keys(n), Vals(n)) for n in range(10)]
sage: D = sage.misc.weak_dict.WeakValueDictionary(L)
```

We remove one dictionary item directly. Another item is removed by means of garbage collection. By consequence, there remain eight items in the dictionary:

```
sage: del D[Keys(2)]
sage: del L[5]
sage: sorted(D.items())
[([0], <0>),
 ([1], <1>),
 ([3], <3>),
 ([4], <4>),
 ([6], <6>),
 ([7], <7>),
 ([8], <8>),
 ([9], <9>)]
```

iteritems()

Iterate over the items of this dictionary.

> **Warning:** Iteration is unsafe, if the length of the dictionary changes during the iteration! This can also happen by garbage collection.

EXAMPLES:

```
sage: import sage.misc.weak_dict
sage: class Vals:
....:     def __init__(self, n):
....:         self.n = n
....:     def __repr__(self):
....:         return "<%s>"%self.n
....:     def __cmp__(self, other):
....:         c = cmp(type(self),type(other))
....:         if c:
....:             return c
....:         return cmp(self.n,other.n)
sage: class Keys(object):
....:     def __init__(self, n):
....:         self.n = n
....:     def __hash__(self):
....:         if self.n%2:
....:             return 5
....:         return 3
....:     def __repr__(self):
....:         return "[%s]"%self.n
....:     def __cmp__(self, other):
....:         c = cmp(type(self),type(other))
....:         if c:
....:             return c
....:         return cmp(self.n,other.n)
sage: L = [(Keys(n), Vals(n)) for n in range(10)]
sage: D = sage.misc.weak_dict.WeakValueDictionary(L)
```

We remove one dictionary item directly. Another item is removed by means of garbage collection. By consequence, there remain eight items in the dictionary:

```
sage: del D[Keys(2)]
sage: del L[5]
sage: for k,v in sorted(D.iteritems()):
....:     print k, v
[0] <0>
[1] <1>
[3] <3>
[4] <4>
[6] <6>
[7] <7>
[8] <8>
[9] <9>
```

iterkeys()

Iterate over the keys of this dictionary.

> **Warning:** Iteration is unsafe, if the length of the dictionary changes during the iteration! This can also happen by garbage collection.

EXAMPLES:

```
sage: import sage.misc.weak_dict
sage: class Vals(object): pass
sage: L = [Vals() for _ in range(10)]
sage: D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))
sage: del L[4]
```

One item got deleted from the list `L` and hence the corresponding item in the dictionary got deleted as well. Therefore, the corresponding key 4 is missing in the list of keys:

```
sage: list(sorted(D.iterkeys()))
[0, 1, 2, 3, 5, 6, 7, 8, 9]
```

itervalues()

Iterate over the values of this dictionary.

> **Warning:** Iteration is unsafe, if the length of the dictionary changes during the iteration! This can also happen by garbage collection.

EXAMPLES:

```
sage: import sage.misc.weak_dict
sage: class Vals:
....:     def __init__(self, n):
....:         self.n = n
....:     def __repr__(self):
....:         return "<%s>"%self.n
....:     def __cmp__(self, other):
....:         c = cmp(type(self),type(other))
....:         if c:
....:             return c
....:         return cmp(self.n,other.n)
sage: L = [Vals(n) for n in range(10)]
sage: D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))
```

We delete one item from `D` and we delete one item from the list `L`. The latter implies that the corresponding item from `D` gets deleted as well. Hence, there remain eight values:

```
sage: del D[2]
sage: del L[5]
sage: for v in sorted(D.itervalues()):
....:     print v
<0>
<1>
<3>
<4>
<6>
<7>
<8>
<9>
```

keys()

The list of keys.

EXAMPLES:

```
sage: import sage.misc.weak_dict
sage: class Vals(object): pass
sage: L = [Vals() for _ in range(10)]
sage: D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))
```

```
sage: del L[4]
```

One item got deleted from the list `L` and hence the corresponding item in the dictionary got deleted as well. Therefore, the corresponding key 4 is missing in the list of keys:

```
sage: sorted(D.keys())
[0, 1, 2, 3, 5, 6, 7, 8, 9]
```

pop(*k*)

Return the value for a given key, and delete it from the dictionary.

EXAMPLES:

```
sage: import sage.misc.weak_dict
sage: L = [GF(p) for p in prime_range(10^3)]
sage: D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))
sage: 20 in D
True
sage: D.pop(20)
Finite Field of size 73
sage: 20 in D
False
sage: D.pop(20)
Traceback (most recent call last):
...
KeyError: 20
```

TESTS:

Check that trac ticket #15956 has been fixed, i.e., a `TypeError` is raised for unhashable objects:

```
sage: D = sage.misc.weak_dict.WeakValueDictionary()
sage: D.pop(matrix([]))
Traceback (most recent call last):
...
TypeError: mutable matrices are unhashable
```

popitem()

Return and delete some item from the dictionary.

EXAMPLES:

```
sage: import sage.misc.weak_dict
sage: D = sage.misc.weak_dict.WeakValueDictionary()
sage: D[1] = ZZ
```

The dictionary only contains a single item, hence, it is clear which one will be returned:

```
sage: D.popitem()
(1, Integer Ring)
```

Now, the dictionary is empty, and hence the next attempt to pop an item will fail with a `KeyError`:

```
sage: D.popitem()
Traceback (most recent call last):
...
KeyError: 'popitem(): weak value dictionary is empty'
```

setdefault(*k*, *default=None*)

Return the stored value for a given key; return and store a default value if no previous value is stored.

EXAMPLES:

```
sage: import sage.misc.weak_dict
sage: L = [(p,GF(p)) for p in prime_range(10)]
sage: D = sage.misc.weak_dict.WeakValueDictionary(L)
sage: len(D)
4
```

The value for an existing key is returned and not overridden:

```
sage: D.setdefault(5, ZZ)
Finite Field of size 5
sage: D[5]
Finite Field of size 5
```

For a non-existing key, the default value is stored and returned:

```
sage: 4 in D
False
sage: D.setdefault(4, ZZ)
Integer Ring
sage: 4 in D
True
sage: D[4]
Integer Ring
sage: len(D)
5
```

TESTS:

Check that trac ticket #15956 has been fixed, i.e., a `TypeError` is raised for unhashable objects:

```
sage: D = sage.misc.weak_dict.WeakValueDictionary()
sage: D.setdefault(matrix([]),ZZ)
Traceback (most recent call last):
...
TypeError: mutable matrices are unhashable
```

values()

Return the list of values.

EXAMPLES:

```
sage: import sage.misc.weak_dict
sage: class Vals:
....:     def __init__(self, n):
....:         self.n = n
....:     def __repr__(self):
....:         return "<%s>"%self.n
....:     def __cmp__(self, other):
....:         c = cmp(type(self),type(other))
....:         if c:
....:             return c
....:         return cmp(self.n,other.n)
sage: L = [Vals(n) for n in range(10)]
sage: D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))
```

We delete one item from `D` and we delete one item from the list `L`. The latter implies that the corresponding item from `D` gets deleted as well. Hence, there remain eight values:

```
sage: del D[2]
sage: del L[5]
```

```
sage: sorted(D.values())
[<0>, <1>, <3>, <4>, <6>, <7>, <8>, <9>]
```

sage.misc.weak_dict.**test_del_dictitem_by_exact_value**(*D*, *value*, *h*)

This function helps testing some cdef function used to delete dictionary items.

INPUT:

- `D` – a Python `<dict>`.
- `value` – an object that is value `D`.
- `h` – the hash of the key under which to find `value` in `D`.

The underlying cdef function deletes an item from `D` that is in the hash bucket determined by `h` and whose value is identic with `value`. Of course, this only makes sense if the pairs `(h, value)` corresponding to items in `D` are pair-wise distinct.

If a matching item can not be found, the function does nothing and silently returns.

TESTS:

See trac ticket #13394 for a discussion.

```
sage: from sage.misc.weak_dict import test_del_dictitem_by_exact_value
sage: B=1000
sage: L=list(range(B))
sage: D1=dict()
sage: D2=dict()
sage: for i in range(100000):        # long time
....:     ki=L[floor(random()*B)]
....:     vi=L[floor(random()*B)]
....:     D1[ki]=vi
....:     D2[ki]=vi
....:     ko=L[floor(random()*B)]
....:     if ko in D1:
....:         vo=D1[ko]
....:         del D1[ko]
....:         test_del_dictitem_by_exact_value(D2,vo,hash(ko))
....:     assert D1 == D2
```

No action is taken if the item prescribed by key hash and value does not exist in the dictionary:

```
sage: D = {1: ZZ}
sage: test_del_dictitem_by_exact_value(D, ZZ, 2)
sage: D
{1: Integer Ring}
sage: test_del_dictitem_by_exact_value(D, QQ, 1)
sage: D
{1: Integer Ring}
```

CHAPTER

SIX

THE C3 ALGORITHM

The C3 algorithm is used as method resolution order for new style classes in Python. The implementation here is used to order the list of super categories of a category.

AUTHOR:

- Simon King (2011-11): initial version.

`sage.misc.c3.`**`C3_algorithm`**(*start*, *bases*, *attribute*, *proper*)

An implementation of the C3 algorithm.

C3 is the algorithm used by Python to construct the method resolution order for new style classes involving multiple inheritance.

After trac ticket #11943 this implementation was used to compute the list of super categories of a category; see `all_super_categories()`. The purpose is to ensure that list of super categories matches with the method resolution order of the parent or element classes of a category.

Since trac ticket #13589, this implementation is superseded by that in `sage.misc.c3_controlled`, that puts the `C3` algorithm under control of some total order on categories. This guarantees that `C3` always finds a consistent Method Resolution Order. For background, see `sage.misc.c3_controlled`.

INPUT:

- `start` – an object; the returned list is built upon data provided by certain attributes of `start`.
- `bases` – a string; the name of an attribute of `start` providing a list of objects.
- `attribute` – a string; the name of an attribute of the objects provided in `getattr(start,bases)`. That attribute is supposed to provide a list.

ASSUMPTIONS:

Our implementation of the algorithm only works on lists of objects that compare equal if and only if they are identical.

OUTPUT:

A list, the result of the C3 algorithm applied to the list `[getattr(X,attribute) for X in getattr(start,bases)]`.

EXAMPLES:

We create a class for elements in a hierarchy that uses the `C3` algorithm to compute, for each element, a linear extension of the elements above it:

```
.. TODO:: Move back the __init__ at the beginning
```

sage: from sage.misc.c3 import C3_algorithm sage: class HierarchyElement(UniqueRepresentation): ....: @lazy_attribute ....: def _all_bases(self): ....: return C3_algorithm(self, '_bases', '_all_bases', False) ....: def __repr__(self): ....: return self._name ....: def __init__(self, name, bases): ....: self._name = name ....: self._bases = list(bases)

We construct a little hierarchy:

```
sage: T = HierarchyElement("T", ())
sage: X = HierarchyElement("X", (T,))
sage: Y = HierarchyElement("Y", (T,))
sage: A = HierarchyElement("A", (X, Y))
sage: B = HierarchyElement("B", (Y, X))
sage: Foo = HierarchyElement("Foo", (A, B))
```

And inspect the linear extensions associated to each element:

```
sage: T._all_bases
[T]
sage: X._all_bases
[X, T]
sage: Y._all_bases
[Y, T]
sage: A._all_bases
[A, X, Y, T]
sage: B._all_bases
[B, Y, X, T]
```

So far so good. However:

```
sage: Foo._all_bases
Traceback (most recent call last):
...
ValueError: Can not merge the items X, Y.
```

The C3 algorithm is not able to create a consistent linear extension. Indeed, its specifications impose that, if X and Y appear in a certain order in the linear extension for an element of the hierarchy, then they should appear in the same order for any lower element. This is clearly not possibly for Foo, since A and B impose incompatible orders. If the above was a hierarchy of classes, Python would complain that it cannot calculate a consistent Method Resolution Order.

TESTS:

Regression test for bug #1 of trac ticket #13501:

```
sage: class C(object): pass
sage: class F(object): pass
sage: class G(object): pass
sage: class B(C,F):    pass
sage: class D(F,G):    pass
sage: class E(F):      pass
sage: class A(B,D,E):  pass
sage: [cls.__name__ for cls in A.mro()]
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'object']

sage: C = HierarchyElement("C", ())
sage: F = HierarchyElement("F", ())
sage: G = HierarchyElement("G", ())
sage: B = HierarchyElement("B", (C, F))
sage: D = HierarchyElement("D", (F, G))
sage: E = HierarchyElement("E", (F,))
sage: A = HierarchyElement("A", (B, D, E))
sage: A._all_bases
[A, B, C, D, E, F, G]
```

Regression test for bug #2 of trac ticket #13501. The following should fail since A asks for B to come before C, where as B is a super class of C:

```
sage: class B(object): pass
sage: class C(B): pass
sage: class A(B, C): pass
Traceback (most recent call last):
...
TypeError: Error when calling the metaclass bases
    Cannot create a consistent method resolution
order (MRO) for bases ...

sage: B = HierarchyElement("B", ())
sage: C = HierarchyElement("C", (B,))
sage: A = HierarchyElement("A", (B,C))
sage: A._all_bases
Traceback (most recent call last):
...
ValueError: Can not merge the items B, C, B.
```

Since trac ticket #11943, the following consistency tests are part of the test suites of categories (except for hom categories):

```
sage: C = Category.join([HopfAlgebrasWithBasis(QQ), FiniteEnumeratedSets()])
sage: C.parent_class.mro() == [x.parent_class for x in C.all_super_categories()]+[object]
True
sage: C.element_class.mro() == [x.element_class for x in C.all_super_categories()]+[object]
True
```

CHAPTER

SEVEN

THE C3 ALGORITHM, UNDER CONTROL OF A TOTAL ORDER.

7.1 Abstract

Python handles multiple inheritance by computing, for each class, a linear extension of the poset of all its super classes (the Method Resolution Order, MRO). The MRO is calculated recursively from local information (the *ordered* list of the direct super classes), with the so-called `C3` algorithm. This algorithm can fail if the local information is not consistent; worst, there exist hierarchies of classes with provably no consistent local information.

For large hierarchy of classes, like those derived from categories in Sage, maintaining consistent local information by hand does not scale and leads to unpredictable `C3` failures (the dreaded "could not find a consistent method resolution order"); a maintenance nightmare.

This module implements a final solution to this problem. Namely, it allows for building automatically the local information from the bare class hierarchy in such a way that guarantees that the `C3` algorithm will never fail.

Err, but you said that this was provably impossible? Well, not if one relaxes a bit the hypotheses; but that's not something one would want to do by hand :-)

7.2 The problem

Consider the following hierarchy of classes:

```
sage: class A1(object): pass
sage: class A2(object):
....:     def foo(self): return 2
sage: class A3(object): pass
sage: class A4(object):
....:     def foo(self): return 4
sage: class A5(A2, A1):
....:     def foo(self): return 5
sage: class A6(A4, A3): pass
sage: class A7(A6, A5): pass
```

If `a` is an instance of `A7`, then Python needs to choose which implementation to use upon calling `a.foo()`: that of `A4` or `A5`, but obviously not that of `A2`. In Python, like in many other dynamic object oriented languages, this is achieved by calculating once for all a specific linear extension of the hierarchy of the super classes of each class, called its Method Resolution Order (MRO):

```
sage: [cls.__name__ for cls in A7.mro()]
['A7', 'A6', 'A4', 'A3', 'A5', 'A2', 'A1', 'object']
```

Thus, in our example, the implementation in `A4` is chosen:

```
sage: a = A7()
sage: a.foo()
4
```

Specifically, the MRO is calculated using the so-called `C3` algorithm which guarantees that the MRO respects not only inheritance, but also the order in which the bases (direct super classes) are given for each class.

However, for large hierarchies of classes with lots of multiple inheritance, like those derived from categories in Sage, this algorithm easily fails if the order of the bases is not chosen consistently (here for `A2` w.r.t. `A1`):

```
sage: class B6(A1,A2): pass
sage: class B7(B6,A5): pass
Traceback (most recent call last):
...
TypeError: Error when calling the metaclass bases
    Cannot create a consistent method resolution
order (MRO) for bases ...
```

There actually exist hierarchies of classes for which `C3` fails whatever order of the bases is chosen; the smallest such example, admittedly artificial, has ten classes (see below). Still, this highlights that this problem has to be tackled in a systematic way.

Fortunately, one can trick `C3`, without changing the inheritance semantic, by adding some super classes of `A` to the bases of `A`. In the following example, we completely force a given MRO by specifying *all* the super classes of `A` as bases:

```
sage: class A7(A6, A5, A4, A3, A2, A1): pass
sage: [cls.__name__ for cls in A7.mro()]
['A7', 'A6', 'A5', 'A4', 'A3', 'A2', 'A1', 'object']
```

Luckily this can be optimized; here it is sufficient to add a single base to enforce the same MRO:

```
sage: class A7(A6, A5, A4): pass
sage: [cls.__name__ for cls in A7.mro()]
['A7', 'A6', 'A5', 'A4', 'A3', 'A2', 'A1', 'object']
```

7.3 A strategy to solve the problem

We should recall at this point a design decision that we took for the hierarchy of classes derived from categories: *the semantic shall only depend on the inheritance order*, not on the specific MRO, and in particular not on the order of the bases (see the section `On the order of super categories` in the `category primer`). If a choice needs to be made (for example for efficiency reasons), then this should be done explicitly, on a method-by-method basis. In practice this design goal is not yet met.

Note: When managing large hierarchies of classes in other contexts this may be too strong a design decision.

The strategy we use for hierarchies of classes derived from categories is then:

1. To choose a global total order on the whole hierarchy of classes.
2. To control `C3` to get it to return MROs that follow this total order.

A basic approach for point 1., that will work for any hierarchy of classes, is to enumerate the classes while they are constructed (making sure that the bases of each class are enumerated before that class), and to order the classes according to that enumeration. A more conceptual ordering may be desirable, in particular to get deterministic and reproducible results. In the context of Sage, this is mostly relevant for those doctests displaying all the categories or classes that an object inherits from.

7.4 Getting fine control on C3

This module is about point 2.

The natural approach would be to change the algorithm used by Python to compute the MRO. However, changing Python's default algorithm just for our needs is obviously not an option, and there is currently no hook to customize specific classes to use a different algorithm. Pushing the addition of such a hook into stock Python would take too much time and effort.

Another approach would be to use the "adding bases" trick straightforwardly, putting the list of *all* the super classes of a class as its bases. However, this would have several drawbacks:

- It is not so elegant, in particular because it duplicates information: we already know through `A5` that `A7` is a subclass of `A1`. This duplication could be acceptable in our context because the hierarchy of classes is generated automatically from a conceptual hierarchy (the categories) which serves as single point of truth for calculating the bases of each class.
- It increases the complexity of the calculation of the MRO with `C3`. For example, for a linear hierachy of classes, the complexity goes from $O(n^2)$ to $O(n^3)$ which is not acceptable.
- It increases the complexity of inspecting the classes. For example, the current implementation of the `dir` command in Python has no cache, and its complexity is linear in the number of maximal paths in the class hierarchy graph as defined by the bases. For a linear hierarchy, this is of complexity $O(p_n)$ where p_n is the number of integer partitions of n, which is exponential. And indeed, running `dir` for a typical class like `GradedHopfAlgebrasWithBasis(QQ).parent_class` with `37` super classes took 18 seconds with this approach.

  Granted: this mostly affects the `dir` command and could be blamed on its current implementation. With appropriate caching, it could be reimplemented to have a complexity roughly linear in the number of classes in the hierarchy. But this won't happen any time soon in a stock Python.

This module refines this approach to make it acceptable, if not seamless. Given a hierarchy and a total order on this hierarchy, it calculates for each element of the hierarchy the smallest list of additional bases that forces `C3` to return the desired MRO. This is achieved by implementing an instrumented variant of the `C3` algorithm (which we call *instrumented ``C3``*) that detects when `C3` is about to take a wrong decision and adds one base to force the right decision. Then, running the standard `C3` algorithm with the updated list of bases (which we call *controlled ``C3``*) yields the desired MRO.

EXAMPLES:

As an experimentation and testing tool, we use a class `HierarchyElement` whose instances can be constructed from a hierarchy described by a poset, a digraph, or more generally a successor relation. By default, the desired MRO is sorted decreasingly. Another total order can be specified using a sorting key.

We consider the smallest poset describing a class hierarchy admitting no MRO whatsoever:

```
sage: P = Poset({10: [9,8,7], 9:[6,1], 8:[5,2], 7:[4,3], 6: [3,2], 5:[3,1], 4: [2,1] }, linear_extens
```

And build a $HierarchyElement$ from it:

```
sage: from sage.misc.c3_controlled import HierarchyElement
sage: x = HierarchyElement(10, P)
```

Here are its bases:

```
sage: HierarchyElement(10, P)._bases
[9, 8, 7]
```

Using the standard C3 algorithm fails:

```
sage: x.mro_standard
Traceback (most recent call last):
...
ValueError: Can not merge the items 3, 3, 2.
```

We also get a failure when we relabel P according to another linear extension. For easy relabelling, we first need to set an appropriate default linear extension for P:

```
sage: linear_extension = list(reversed(IntegerRange(1,11)))
sage: P = P.with_linear_extension(linear_extension)
sage: list(P)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Now, we play with the fifth linear extension of P:

```
sage: L = P.linear_extensions()
sage: Q = L[5].to_poset()
sage: Q.cover_relations()
[[10, 9], [10, 8], [10, 7], [9, 6], [9, 3], [8, 5], [8, 2], [7, 4], [7, 1], [6, 2], [6, 1], [5, 3],
sage: x = HierarchyElement(10, Q)
sage: x.mro_standard
Traceback (most recent call last):
...
ValueError: Can not merge the items 2, 3, 3.
```

On the other hand, both the instrumented C3 algorithm, and the controlled C3 algorithm give the desired MRO:

```
sage: x.mro
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
sage: x.mro_controlled
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

The above checks, and more, can be run with:

```
sage: x._test_mro()
```

In practice, the control was achieved by adding the following bases:

```
sage: x._bases
[9, 8, 7]
sage: x._bases_controlled
[9, 8, 7, 6, 5]
```

Altogether, four bases were added for control:

```
sage: sum(len(HierarchyElement(q, Q)._bases) for q in Q)
15
sage: sum(len(HierarchyElement(q, Q)._bases_controlled) for q in Q)
19
```

This information can also be recovered with:

```
sage: x.all_bases_len()
15
sage: x.all_bases_controlled_len()
19
```

We now check that the C3 algorithm fails for all linear extensions l of this poset, whereas both the instrumented and controlled C3 algorithms succeed; along the way, we collect some statistics:

```
sage: stats = []
sage: for l in L:
....:     x = HierarchyElement(10, l.to_poset())
....:     try: # Check that x.mro_standard always fails with a ValueError
....:         x.mro_standard
....:     except ValueError:
....:         pass
....:     else:
....:         assert False
....:     assert x.mro            == list(P)
....:     assert x.mro_controlled == list(P)
....:     assert x.all_bases_len() == 15
....:     stats.append(x.all_bases_controlled_len()-x.all_bases_len())
```

Depending on the linear extension l it was necessary to add between one and five bases for control; for example, 216 linear extensions required the addition of four bases:

```
sage: Word(stats).evaluation_sparse()
[(1, 36), (2, 108), (3, 180), (4, 216), (5, 180)]
```

We now consider a hierarchy of categories:

```
sage: from operator import attrgetter
sage: x = HierarchyElement(Groups(), attrcall("super_categories"), attrgetter("_cmp_key"))
sage: x.mro
[Category of groups, Category of monoids, Category of semigroups,
 Category of inverse unital magmas, Category of unital magmas, Category of magmas,
 Category of sets, Category of sets with partial maps, Category of objects]
sage: x.mro_standard
[Category of groups, Category of monoids, Category of semigroups,
 Category of inverse unital magmas, Category of unital magmas, Category of magmas,
 Category of sets, Category of sets with partial maps, Category of objects]
```

For a typical category, few bases, if any, need to be added to force C3 to give the desired order:

```
sage: C = FiniteFields()
sage: x = HierarchyElement(C, attrcall("super_categories"), attrgetter("_cmp_key"))
sage: x.mro == x.mro_standard
False
sage: x.all_bases_len()
66
sage: x.all_bases_controlled_len()
69

sage: C = GradedHopfAlgebrasWithBasis(QQ)
sage: x = HierarchyElement(C, attrcall("super_categories"), attrgetter("_cmp_key"))
sage: x._test_mro()
sage: x.mro == x.mro_standard
False
sage: x.all_bases_len()
```

```
82
sage: x.all_bases_controlled_len()
89
```

The following can be used to search through the Sage named categories for any that requires the addition of some bases:

```
sage: from sage.categories.category import category_sample
sage: sorted([C for C in category_sample()
....:          if len(C._super_categories_for_classes) != len(C.super_categories())],
....:         key=str)
[Category of affine weyl groups,
 Category of fields,
 Category of finite dimensional algebras with basis over Rational Field,
 Category of finite dimensional hopf algebras with basis over Rational Field,
 Category of graded hopf algebras with basis over Rational Field,
 Category of hopf algebras with basis over Rational Field]
```

AUTHOR:

- Nicolas M. Thiery (2012-09): initial version.

`sage.misc.c3_controlled.`**`C3_merge`**(*lists*)

Return the input lists merged using the `C3` algorithm.

EXAMPLES:

```
sage: from sage.misc.c3_controlled import C3_merge
sage: C3_merge([[3,2],[4,3,1]])
[4, 3, 2, 1]
sage: C3_merge([[3,2],[4,1]])
[3, 2, 4, 1]
```

This function is only used for testing and experimenting purposes, but exercised quite some by the other doctests in this file.

It is an extract of `sage.misc.c3.C3_algorithm()`; the latter could be possibly rewritten to use this one to avoid duplication.

`sage.misc.c3_controlled.`**`C3_sorted_merge`**(*lists*, *key='identity'*)

Return the sorted input lists merged using the `C3` algorithm, with a twist.

INPUT:

- `lists` – a non empty list (or iterable) of lists (or iterables), each sorted strictly decreasingly according to `key`
- `key` – a function

OUTPUT: a pair `(result, suggestion)`

`result` is the sorted list obtained by merging the lists in `lists` while removing duplicates, and `suggestion` is a list such that applying `C3` on `lists` with its last list replaced by `suggestion` would return `result`.

EXAMPLES:

With the following input, `C3_merge()` returns right away a sorted list:

```
sage: from sage.misc.c3_controlled import C3_merge
sage: C3_merge([[2],[1]])
[2, 1]
```

In that case, `C3_sorted_merge()` returns the same result, with the last line unchanged:

```
sage: from sage.misc.c3_controlled import C3_sorted_merge
sage: C3_sorted_merge([[2],[1]])
([2, 1], [1])
```

On the other hand, with the following input, `C3_merge()` returns a non sorted list:

```
sage: C3_merge([[1],[2]])
[1, 2]
```

Then, `C3_sorted_merge()` returns a sorted list, and suggests to replace the last line by `[2,1]`:

```
sage: C3_sorted_merge([[1],[2]])
([2, 1], [2, 1])
```

And indeed `C3_merge` now returns the desired result:

```
sage: C3_merge([[1],[2,1]])
[2, 1]
```

From now on, we use this little wrapper that checks that `C3_merge`, with the suggestion of `C3_sorted_merge`, returns a sorted list:

```
sage: def C3_sorted_merge_check(lists):
....:     result, suggestion = C3_sorted_merge(lists)
....:     assert result == C3_merge(lists[:-1] + [suggestion])
....:     return result, suggestion
```

Base cases:

```
sage: C3_sorted_merge_check([])
Traceback (most recent call last):
...
ValueError: The input should be a non empty list of lists (or iterables)
sage: C3_sorted_merge_check([[]])
([], [])
sage: C3_sorted_merge_check([[1]])
([1], [1])
sage: C3_sorted_merge_check([[3,2,1]])
([3, 2, 1], [3, 2, 1])
sage: C3_sorted_merge_check([[1],[1]])
([1], [1])
sage: C3_sorted_merge_check([[3,2,1],[3,2,1]])
([3, 2, 1], [3, 2, 1])
```

Exercise different states for the last line:

```
sage: C3_sorted_merge_check([[1],[2],[]])
([2, 1], [2, 1])
sage: C3_sorted_merge_check([[1],[2], [1]])
([2, 1], [2, 1])
```

Explore (all?) the different execution branches:

```
sage: C3_sorted_merge_check([[3,1],[4,2]])
([4, 3, 2, 1], [4, 3, 2, 1])
sage: C3_sorted_merge_check([[4,1],[3,2]])
([4, 3, 2, 1], [3, 2, 1])
sage: C3_sorted_merge_check([[3,2],[4,1]])
([4, 3, 2, 1], [4, 3, 1])
sage: C3_sorted_merge_check([[1],[4,3,2]])
```

```
([4, 3, 2, 1], [4, 3, 2, 1])
sage: C3_sorted_merge_check([[1],[3,2], []])
([3, 2, 1], [2, 1])
sage: C3_sorted_merge_check([[1],[4,3,2], []])
([4, 3, 2, 1], [2, 1])
sage: C3_sorted_merge_check([[1],[4,3,2], [2]])
([4, 3, 2, 1], [2, 1])
sage: C3_sorted_merge_check([[2],[1],[4],[3]])
([4, 3, 2, 1], [3, 2, 1])
sage: C3_sorted_merge_check([[2],[1],[4],[]])
([4, 2, 1], [4, 2, 1])
sage: C3_sorted_merge_check([[2],[1],[3],[4]])
([4, 3, 2, 1], [4, 3, 2, 1])
sage: C3_sorted_merge_check([[2],[1],[3,2,1],[3]])
([3, 2, 1], [3])
sage: C3_sorted_merge_check([[2],[1],[2,1],[3]])
([3, 2, 1], [3, 2])
```

Exercises adding one item when the last list has a single element; the second example comes from an actual poset:

```
sage: C3_sorted_merge_check([[5,4,2],[4,3],[5,4,1]])
([5, 4, 3, 2, 1], [5, 4, 3, 2, 1])
sage: C3_sorted_merge_check([[6,4,2],[5,3],[6,5,1]])
([6, 5, 4, 3, 2, 1], [6, 5, 4, 3, 2, 1])
```

class `sage.misc.c3_controlled.`**`CmpKey`**

Bases: `object`

This class implements the lazy attribute `Category._cmp_key`.

The comparison key `A._cmp_key` of a category is used to define an (almost) total order on non-join categories by setting, for two categories A and B, $A < B$ if `A._cmp_key > B._cmp_key`. This order in turn is used to give a normal form to join's, and help toward having a consistent method resolution order for parent/element classes.

The comparison key should satisfy the following properties:

- If A is a subcategory of B, then $A < B$ (so that `A._cmp_key > B._cmp_key`). In particular, `Objects()` is the largest category.
- If $A! = B$ and taking the join of A and B makes sense (e.g. taking the join of `Algebras(GF(5))` and `Algebras(QQ)` does not make sense), then $A < B$ or $B < A$.

The rationale for the inversion above between $A < B$ and `A._cmp_key > B._cmp_key` is that we want the order to be compatible with inclusion of categories, yet it's easier in practice to create keys that get bigger and bigger while we go down the category hierarchy.

This implementation applies to join-irreducible categories (i.e. categories that are not join categories). It returns a pair of integers `(flags, i)`, where `flags` is to be interpreted as a bit vector. The first bit is set if `self` is a facade set. The second bit is set if `self` is finite. And so on. The choice of the flags is adhoc and was primarily crafted so that the order between categories would not change too much upon integration of trac ticket #13589 and would be reasonably session independent. The number `i` is there to resolve ambiguities; it is session dependent, and is assigned increasingly when new categories are created.

Note: This is currently not implemented using a `lazy_attribute` for speed reasons only (the code is in Cython and takes advantage of the fact that Category objects always have a `__dict__` dictionary)

Todo

- Handle nicely (covariant) functorial constructions and axioms

EXAMPLES:

```
sage: Objects()._cmp_key
(0, 0)
sage: SetsWithPartialMaps()._cmp_key
(0, 1)
sage: Sets()._cmp_key
(0, 2)
sage: Sets().Facade()._cmp_key
(1, ...)
sage: Sets().Finite()._cmp_key
(2, ...)
sage: Sets().Infinite()._cmp_key
(4, ...)
sage: EnumeratedSets()._cmp_key
(8, ...)
sage: FiniteEnumeratedSets()._cmp_key
(10, ...)
sage: SetsWithGrading()._cmp_key
(16, ...)
sage: Posets()._cmp_key
(32, ...)
sage: LatticePosets()._cmp_key
(96, ...)
sage: Crystals()._cmp_key
(136, ...)
sage: AdditiveMagmas()._cmp_key
(256, ...)
sage: Magmas()._cmp_key
(4096, ...)
sage: CommutativeAdditiveSemigroups()._cmp_key
(256, ...)
sage: Rings()._cmp_key
(225536, ...)
sage: Algebras(QQ)._cmp_key
(225536, ...)
sage: AlgebrasWithBasis(QQ)._cmp_key
(227584, ...)
sage: GradedAlgebras(QQ)._cmp_key
(226560, ...)
sage: GradedAlgebrasWithBasis(QQ)._cmp_key
(228608, ...)
```

For backward compatibility we currently want the following comparisons:

```
sage: EnumeratedSets()._cmp_key > Sets().Facade()._cmp_key
True
sage: AdditiveMagmas()._cmp_key > EnumeratedSets()._cmp_key
True

sage: Category.join([EnumeratedSets(), Sets().Facade()]).parent_class._an_element_.__module__
'sage.categories.enumerated_sets'

sage: CommutativeAdditiveSemigroups()._cmp_key < Magmas()._cmp_key
True
sage: VectorSpaces(QQ)._cmp_key < Rings()._cmp_key
```

```
True
sage: VectorSpaces(QQ)._cmp_key < Magmas()._cmp_key
True
```

class sage.misc.c3_controlled.**CmpKeyNamed**

Bases: object

This class implements the lazy attribute CategoryWithParameters._cmp_key.

See also:

- CmpKey
- lazy_attribute
- sage.categories.category.CategoryWithParameters.

Note:

- The value of the attribute depends only on the parameters of this category.
- This is currently not implemented using a lazy_attribute for speed reasons only.

EXAMPLES:

```
sage: Algebras(GF(3))._cmp_key == Algebras(GF(5))._cmp_key  # indirect doctest
True
sage: Algebras(ZZ)._cmp_key != Algebras(GF(5))._cmp_key
True
```

class sage.misc.c3_controlled.**HierarchyElement**(*value*, *bases*, *key*, *from_value*)

Bases: object

A class for elements in a hierarchy.

This class is for testing and experimenting with various variants of the C3 algorithm to compute a linear extension of the elements above an element in a hierarchy. Given the topic at hand, we use the following naming conventions. For x an element of the hierarchy, we call the elements just above x its *bases*, and the linear extension of all elements above x its *MRO*.

By convention, the bases are given as lists of HierarchyElement s, and MROs are given a list of the corresponding values.

INPUT:

- value – an object
- succ – a successor function, poset or digraph from which one can recover the successors of value
- key – a function taking values as input (default: the identity) this function is used to compute comparison keys for sorting elements of the hierarchy.

Note: Constructing a HierarchyElement immediately constructs the whole hierarchy above it.

EXAMPLES:

See the introduction of this module sage.misc.c3_controlled for many examples. Here we consider a large example, originaly taken from the hierarchy of categories above HopfAlgebrasWithBasis:

```
sage: from sage.misc.c3_controlled import HierarchyElement
sage: G = DiGraph({
....:     44 :  [43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24,
```

```
....:     43 :  [42, 41, 40, 36, 35, 39, 38, 37, 33, 32, 31, 30, 29, 28, 27, 26, 23, 22, 21, 20,
....:     42 :  [36, 35, 37, 30, 29, 28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
....:     41 :  [40, 36, 35, 33, 32, 31, 30, 29, 28, 27, 26, 23, 22, 21, 20, 19, 18, 17, 16, 15,
....:     40 :  [36, 35, 32, 31, 30, 29, 28, 27, 26, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9,
....:     39 :  [38, 37, 33, 32, 31, 30, 29, 28, 27, 26, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14,
....:     38 :  [37, 33, 32, 31, 30, 29, 28, 27, 26, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13,
....:     37 :  [30, 29, 28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
....:     36 :  [35, 30, 29, 28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
....:     35 :  [29, 28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
....:     34 :  [33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14,
....:     33 :  [32, 31, 30, 29, 28, 27, 26, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11,
....:     32 :  [31, 30, 29, 28, 27, 26, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5,
....:     31 :  [30, 29, 28, 27, 26, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
....:     30 :  [29, 28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
....:     29 :  [28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
....:     28 :  [27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
....:     27 :  [15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
....:     26 :  [15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
....:     25 :  [24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3
....:     24 :  [4, 2, 1, 0],
....:     23 :  [22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
....:     22 :  [21, 20, 18, 17, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
....:     21 :  [20, 17, 4, 2, 1, 0],
....:     20 :  [4, 2, 1, 0],
....:     19 :  [18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
....:     18 :  [17, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
....:     17 :  [4, 2, 1, 0],
....:     16 :  [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
....:     15 :  [14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
....:     14 :  [11, 3, 2, 1, 0],
....:     13 :  [12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
....:     12 :  [11, 9, 8, 5, 3, 2, 1, 0],
....:     11 :  [3, 2, 1, 0],
....:     10 :  [9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
....:     9 :  [8, 5, 3, 2, 1, 0],
....:     8 :  [3, 2, 1, 0],
....:     7 :  [6, 5, 4, 3, 2, 1, 0],
....:     6 :  [4, 3, 2, 1, 0],
....:     5 :  [3, 2, 1, 0],
....:     4 :  [2, 1, 0],
....:     3 :  [2, 1, 0],
....:     2 :  [1, 0],
....:     1 :  [0],
....:     0 :  [],
....:     })

sage: x = HierarchyElement(44, G)
sage: x.mro
[44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21,
sage: x.cls
<class '44.cls'>
sage: x.cls.mro()
[<class '44.cls'>, <class '43.cls'>, <class '42.cls'>, <class '41.cls'>, <class '40.cls'>, <clas
```

all_bases()

Return the set of all the `HierarchyElement``s above ``self`, `self` included.

EXAMPLES:

```
sage: from sage.misc.c3_controlled import HierarchyElement
sage: P = Poset((divisors(30), lambda x,y: y.divides(x)), facade=True)
sage: HierarchyElement(1, P).all_bases()
{1}
sage: HierarchyElement(10, P).all_bases()  # random output
{10, 5, 2, 1}
sage: sorted([x.value for x in HierarchyElement(10, P).all_bases()])
[1, 2, 5, 10]
```

all_bases_controlled_len()

Return the cumulated size of the controlled bases of the elements above `self` in the hierarchy.

EXAMPLES:

```
sage: from sage.misc.c3_controlled import HierarchyElement
sage: P = Poset((divisors(30), lambda x,y: y.divides(x)), facade=True)
sage: HierarchyElement(30, P).all_bases_controlled_len()
13
```

all_bases_len()

Return the cumulated size of the bases of the elements above `self` in the hierarchy.

EXAMPLES:

```
sage: from sage.misc.c3_controlled import HierarchyElement
sage: P = Poset((divisors(30), lambda x,y: y.divides(x)), facade=True)
sage: HierarchyElement(30, P).all_bases_len()
12
```

bases()

The bases of `self`.

The bases are given as a list of `HierarchyElement``s, sorted decreasingly accoding to the ``key` function.

EXAMPLES:

```
sage: from sage.misc.c3_controlled import HierarchyElement
sage: P = Poset((divisors(30), lambda x,y: y.divides(x)), facade=True)
sage: x = HierarchyElement(10, P)
sage: x.bases
[5, 2]
sage: type(x.bases[0])
<class 'sage.misc.c3_controlled.HierarchyElement'>
sage: x.mro
[10, 5, 2, 1]
sage: x._bases_controlled
[5, 2]
```

cls()

Return a Python class with inheritance graph parallel to the hierarchy above `self`.

EXAMPLES:

```
sage: from sage.misc.c3_controlled import HierarchyElement
sage: P = Poset((divisors(30), lambda x,y: y.divides(x)), facade=True)
sage: x = HierarchyElement(1, P)
sage: x.cls
<class '1.cls'>
sage: x.cls.mro()
[<class '1.cls'>, <type 'object'>]
```

```
sage: x = HierarchyElement(30, P)
sage: x.cls
<class '30.cls'>
sage: x.cls.mro()
[<class '30.cls'>, <class '15.cls'>, <class '10.cls'>, <class '6.cls'>, <class '5.cls'>, <cl
```

mro()

The MRO for this object, calculated with `C3_sorted_merge()`.

EXAMPLES:

```
sage: from sage.misc.c3_controlled import HierarchyElement, C3_sorted_merge, identity
sage: P = Poset({7: [5,6], 5:[1,2], 6: [3,4]}, facade = True)
sage: x = HierarchyElement(5, P)
sage: x.mro
[5, 2, 1]
sage: x = HierarchyElement(6, P)
sage: x.mro
[6, 4, 3]
sage: x = HierarchyElement(7, P)
sage: x.mro
[7, 6, 5, 4, 3, 2, 1]

sage: C3_sorted_merge([[6, 4, 3], [5, 2, 1], [6, 5]], identity)
([6, 5, 4, 3, 2, 1], [6, 5, 4])
```

TESTS:

```
sage: assert all(isinstance(v, Integer) for v in x.mro)
```

mro_controlled()

The MRO for this object, calculated with `C3_merge()`, under control of $C3_sorted_merge$

EXAMPLES:

```
sage: from sage.misc.c3_controlled import HierarchyElement, C3_merge
sage: P = Poset({7: [5,6], 5:[1,2], 6: [3,4]}, facade=True)
sage: x = HierarchyElement(5, P)
sage: x.mro_controlled
[5, 2, 1]
sage: x = HierarchyElement(6, P)
sage: x.mro_controlled
[6, 4, 3]
sage: x = HierarchyElement(7, P)
sage: x.mro_controlled
[7, 6, 5, 4, 3, 2, 1]
sage: x._bases
[6, 5]
sage: x._bases_controlled
[6, 5, 4]
sage: C3_merge([[6, 4, 3], [5, 2, 1], [6, 5]])
[6, 4, 3, 5, 2, 1]
sage: C3_merge([[6, 4, 3], [5, 2, 1], [6, 5, 4]])
[6, 5, 4, 3, 2, 1]
```

TESTS:

```
sage: assert all(isinstance(v, Integer) for v in x.mro_controlled)
```

mro_standard()

The MRO for this object, calculated with C3_merge()

EXAMPLES:

```
sage: from sage.misc.c3_controlled import HierarchyElement, C3_merge
sage: P = Poset({7: [5,6], 5:[1,2], 6: [3,4]}, facade=True)
sage: x = HierarchyElement(5, P)
sage: x.mro_standard
[5, 2, 1]
sage: x = HierarchyElement(6, P)
sage: x.mro_standard
[6, 4, 3]
sage: x = HierarchyElement(7, P)
sage: x.mro_standard
[7, 6, 4, 3, 5, 2, 1]
sage: C3_merge([[6, 4, 3], [5, 2, 1], [6, 5]])
[6, 4, 3, 5, 2, 1]
```

TESTS:

```
sage: assert all(isinstance(v, Integer) for v in x.mro_standard)
```

sage.misc.c3_controlled.**identity**(*x*)

EXAMPLES:

```
sage: from sage.misc.c3_controlled import identity
sage: identity(10)
10
```

CHAPTER

EIGHT

DECORATORS

Python decorators for use in Sage.

AUTHORS:

- Tim Dumol (5 Dec 2009) – initial version.
- Johan S. R. Nielsen (2010) – collect decorators from various modules.
- Johan S. R. Nielsen (8 apr 2011) – improve introspection on decorators.
- Simon King (2011-05-26) – improve introspection of sage_wraps. Put this file into the reference manual.
- Julian Rueth (2014-03-19): added `decorator_keywords` decorator

`sage.misc.decorators.`**`decorator_defaults`**(*func*)

This function allows a decorator to have default arguments.

Normally, a decorator can be called with or without arguments. However, the two cases call for different types of return values. If a decorator is called with no parentheses, it should be run directly on the function. However, if a decorator is called with parentheses (i.e., arguments), then it should return a function that is then in turn called with the defined function as an argument.

This decorator allows us to have these default arguments without worrying about the return type.

EXAMPLES:

```
sage: from sage.misc.decorators import decorator_defaults
sage: @decorator_defaults
... def my_decorator(f,*args,**kwds):
...     print kwds
...     print args
...     print f.__name__
...
sage: @my_decorator
... def my_fun(a,b):
...     return a,b
...
{}
()
my_fun
sage: @my_decorator(3,4,c=1,d=2)
... def my_fun(a,b):
...     return a,b
...
{'c': 1, 'd': 2}
(3, 4)
my_fun
```

sage.misc.decorators.**decorator_keywords**(*func*)

A decorator for decorators with optional keyword arguments.

EXAMPLES:

```
sage: from sage.misc.decorators import decorator_keywords
sage: @decorator_keywords
....: def preprocess(f=None, processor=None):
....:     def wrapper(*args, **kwargs):
....:         if processor is not None:
....:             args, kwargs = processor(*args, **kwargs)
....:         return f(*args, **kwargs)
....:     return wrapper
```

This decorator can be called with and without arguments:

```
sage: @preprocess
....: def foo(x): return x
sage: foo(None)
sage: foo(1)
1
```

```
sage: def normalize(x): return ((0,),{}) if x is None else ((x,),{})
sage: @preprocess(processor=normalize)
....: def foo(x): return x
sage: foo(None)
0
sage: foo(1)
1
```

class sage.misc.decorators.**infix_operator**(*precedence*)

Bases: object

A decorator for functions which allows for a hack that makes the function behave like an infix operator.

This decorator exists as a convenience for interactive use.

EXAMPLES:

An infix dot product operator:

```
sage: def dot(a,b): return a.dot_product(b)
sage: dot=infix_operator('multiply')(dot)
sage: u=vector([1,2,3])
sage: v=vector([5,4,3])
sage: u *dot* v
22
```

An infix element-wise addition operator:

```
sage: def eadd(a,b):
...     return a.parent([i+j for i,j in zip(a,b)])
sage: eadd=infix_operator('add')(eadd)
sage: u=vector([1,2,3])
sage: v=vector([5,4,3])
sage: u +eadd+ v
(6, 6, 6)
sage: 2*u +eadd+ v
(7, 8, 9)
```

A hack to simulate a postfix operator:

```
sage: def thendo(a,b): return b(a)
sage: thendo=infix_operator('or')(thendo)
sage: x |thendo| cos |thendo| (lambda x: x^2)
cos(x)^2
```

class sage.misc.decorators.**options**(***options*)

Bases: object

A decorator for functions which allows for default options to be set and reset by the end user. Additionally, if one needs to, one can get at the original keyword arguments passed into the decorator.

TESTS:

```
sage: from sage.misc.decorators import options
sage: o = options(rgbcolor=(0,0,1))
sage: o.options
{'rgbcolor': (0, 0, 1)}
sage: o = options(rgbcolor=(0,0,1), __original_opts=True)
sage: o.original_opts
True
sage: loads(dumps(o)).options
{'rgbcolor': (0, 0, 1)}
```

Demonstrate that the introspected argument specification of the wrapped function is updated (see trac ticket #9976):

```
sage: from sage.misc.decorators import options
sage: o = options(rgbcolor=(0,0,1))
sage: def f(*args, **kwds): print args, list(sorted(kwds.items()))
sage: f1 = o(f)
sage: from sage.misc.sageinspect import sage_getargspec
sage: sage_getargspec(f1)
ArgSpec(args=['rgbcolor'], varargs='args', keywords='kwds', defaults=((0, 0, 1),))
```

class sage.misc.decorators.**rename_keyword**(*deprecated=None*, *deprecation=None*, ***renames*)

Bases: object

A decorator which renames keyword arguments and optionally deprecates the new keyword.

INPUT:

- deprecation – integer. The trac ticket number where the deprecation was introduced.
- the rest of the arguments is a list of keyword arguments in the form renamed_option='existing_option'. This will have the effect of renaming renamed_option so that the function only sees existing_option. If both renamed_option and existing_option are passed to the function, existing_option will override the renamed_option value.

EXAMPLES:

```
sage: from sage.misc.decorators import rename_keyword
sage: r = rename_keyword(color='rgbcolor')
sage: r.renames
{'color': 'rgbcolor'}
sage: loads(dumps(r)).renames
{'color': 'rgbcolor'}
```

To deprecate an old keyword:

```
sage: r = rename_keyword(deprecation=13109, color='rgbcolor')
```

sage.misc.decorators.**sage_wraps**(*wrapped*, *assigned=('__module__', '__name__', '__doc__')*, *updated=('__dict__',)*)

Decorator factory which should be used in decorators for making sure that meta-information on the decorated callables are retained through the decorator, such that the introspection functions of sage.misc.sageinspect retrieves them correctly. This includes documentation string, source, and argument specification. This is an extension of the Python standard library decorator functools.wraps.

That the argument specification is retained from the decorated functions implies, that if one uses sage_wraps in a decorator which intentionally changes the argument specification, one should add this information to the special attribute _sage_argspec_ of the wrapping function (for an example, see e.g. @options decorator in this module).

EXAMPLES:

Demonstrate that documentation string and source are retained from the decorated function:

```
sage: def square(f):
...     @sage_wraps(f)
...     def new_f(x):
...         return f(x)*f(x)
...     return new_f
sage: @square
... def g(x):
...     "My little function"
...     return x
sage: g(2)
4
sage: g(x)
x^2
sage: g.__doc__
'My little function'
sage: from sage.misc.sageinspect import sage_getsource, sage_getsourcelines, sage_getfile
sage: sage_getsource(g)
'@square...def g(x)...'
```

Demonstrate that the argument description are retained from the decorated function through the special method (when left unchanged) (see trac ticket #9976):

```
sage: def diff_arg_dec(f):
...     @sage_wraps(f)
...     def new_f(y, some_def_arg=2):
...         return f(y+some_def_arg)
...     return new_f
sage: @diff_arg_dec
... def g(x):
...     return x
sage: g(1)
3
sage: g(1, some_def_arg=4)
5
sage: from sage.misc.sageinspect import sage_getargspec
sage: sage_getargspec(g)
ArgSpec(args=['x'], varargs=None, keywords=None, defaults=None)
```

Demonstrate that it correctly gets the source lines and the source file, which is essential for interactive code edition; note that we do not test the line numbers, as they may easily change:

```
sage: P.<x,y> = QQ[]
sage: I = P*[x,y]
sage: sage_getfile(I.interreduced_basis)
'.../sage/interfaces/singular.py'
sage: sage_getsourcelines(I.interreduced_basis)
(['    @singular_gb_standard_options\n',
  '    @libsingular_gb_standard_options\n',
  '    def interreduced_basis(self):\n',
  ...
  '        return self.basis.reduced()\n'], ...)
```

Demonstrate that sage_wraps works for non-function callables (trac ticket #9919):

```
sage: def square_for_met(f):
...       @sage_wraps(f)
...       def new_f(self, x):
...           return f(self,x)*f(self,x)
...       return new_f
sage: class T:
...       @square_for_met
...       def g(self, x):
...           "My little method"
...           return x
sage: t = T()
sage: t.g(2)
4
sage: t.g.__doc__
'My little method'
```

The bug described in trac ticket #11734 is fixed:

```
sage: def square(f):
...       @sage_wraps(f)
...       def new_f(x):
...           return f(x)*f(x)
...       return new_f
sage: f = lambda x:x^2
sage: g = square(f)
sage: g(3) # this line used to fail for some people if these command were manually entered on th
81
```

class sage.misc.decorators.**specialize**(**args*, ***kwargs*)

A decorator generator that returns a decorator that in turn returns a specialized function for function `f`. In other words, it returns a function that acts like `f` with arguments `*args` and `**kwargs` supplied.

INPUT:

- `*args`, `**kwargs` – arguments to specialize the function for.

OUTPUT:

- a decorator that accepts a function `f` and specializes it with `*args` and `**kwargs`

EXAMPLES:

```
sage: f = specialize(5)(lambda x, y: x+y)
sage: f(10)
15
sage: f(5)
10
sage: @specialize("Bon Voyage")
```

```
... def greet(greeting, name):
...     print "{0}, {1}!".format(greeting, name)
sage: greet("Monsieur Jean Valjean")
Bon Voyage, Monsieur Jean Valjean!
sage: greet(name = 'Javert')
Bon Voyage, Javert!
```

class sage.misc.decorators.**suboptions**(*name*, ***options*)
Bases: object

A decorator for functions which collects all keywords starting with name+'_' and collects them into a dictionary which will be passed on to the wrapped function as a dictionary called name_options.

The keyword arguments passed into the constructor are taken to be default for the name_options dictionary.

EXAMPLES:

```
sage: from sage.misc.decorators import suboptions
sage: s = suboptions('arrow', size=2)
sage: s.name
'arrow_'
sage: s.options
{'size': 2}
```

CHAPTER

NINE

LAZY LISTS

A lazy list is an iterator that behaves like a list and possesses a cache mechanism. A lazy list is potentially infinite and speed performances of the cache is comparable with Python lists. One major difference with original Python list is that lazy list are immutable. The advantage is that slices share memory.

EXAMPLES:

```
sage: from sage.misc.lazy_list import lazy_list
sage: P = lazy_list(Primes())
sage: P[100]
547
sage: P[10:34]
lazy list [31, 37, 41, ...]
sage: P[12:23].list()
[41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83]
```

```
sage: f = lazy_list((i**2-3*i for i in xrange(10)))
sage: for i in f: print i,
0 -2 -2 0 4 10 18 28 40 54
sage: i1 = iter(f)
sage: i2 = iter(f)
sage: print i1.next(), i1.next()
0 -2
sage: print i2.next(), i2.next()
0 -2
sage: print i1.next(), i2.next()
-2 -2
```

It is possible to prepend a list to a lazy list:

```
sage: from itertools import count
sage: l = [3,7] + lazy_list(i**2 for i in count())
sage: l
lazy list [3, 7, 0, ...]
```

But, naturally, not the other way around:

```
sage: lazy_list(i-1 for i in count()) + [3,2,5]
Traceback (most recent call last):
...
TypeError: can only add list to lazy_list
```

class sage.misc.lazy_list.**lazy_list**

Bases: object

Lazy list.

INPUT:

- `iterator` – an iterable or an iterator
- `cache` – `None` (default) or a list - used to initialize the cache.
- `start`, `stop`, `step` – `None` (default) or a non-negative integer - parameters for slices

EXAMPLES:

```
sage: from sage.misc.lazy_list import lazy_list
sage: from itertools import count
sage: m = lazy_list(count()); m
lazy list [0, 1, 2, ...]

sage: m2 = lazy_list(count(), start=8, stop=20551, step=2)
sage: m2
lazy list [8, 10, 12, ...]

sage: x = iter(m)
sage: print x.next(), x.next(), x.next()
0 1 2
sage: y = iter(m)
sage: print y.next(), y.next(), y.next()
0 1 2
sage: print x.next(), y.next()
3 3
sage: loads(dumps(m))
lazy list [0, 1, 2, ...]
```

Note:

- `lazy_list` interprets the constant `(size_t)-1` as infinity
- all entry indices are stictly less than `stop` so that `lazy_list` agrees with `range(start, stop)`

get (*i*)

Return the element at position `i`.

If the index is not an integer, then raise a `TypeError`. If the argument is negative then raise a `ValueError`. Finally, if the argument is beyond the size of that lazy list it raises a `IndexError`.

EXAMPLES:

```
sage: from sage.misc.lazy_list import lazy_list
sage: from itertools import chain, repeat
sage: f = lazy_list(chain(iter([1,2,3]), repeat('a')))
sage: f.get(0)
1
sage: f.get(3)
'a'
sage: f.get(0)
1
sage: f.get(4)
'a'

sage: g = f[:10]
sage: g.get(5)
'a'
sage: g.get(10)
Traceback (most recent call last):
```

```
...
IndexError: lazy list index out of range
sage: g.get(1/2)
Traceback (most recent call last):
...
TypeError: rational is not an integer
```

info()

Print information about `self` on standard output.

EXAMPLES:

```
sage: from sage.misc.lazy_list import lazy_list
sage: P = lazy_list(iter(Primes()))[10:21474838:4]
sage: P.info()
cache length 0
start        10
stop         21474838
step         4
sage: P[0]
31
sage: P.info()
cache length 11
start        10
stop         21474838
step         4
```

list()

Return the list made of the elements of `self`.

Note: If the iterator is sufficiently large, this will build a list of length `(size_t)-1` which should be beyond the capacity of your RAM!

EXAMPLES:

```
sage: from sage.misc.lazy_list import lazy_list
sage: P = lazy_list(iter(Primes()))
sage: P[2:143:5].list()
[5, 19, 41, 61, 83, 107, 137, 163, 191, 223, 241, 271, 307, 337, 367, 397, 431, 457, 487, 52
sage: P = lazy_list(iter([1,2,3]))
sage: P.list()
[1, 2, 3]
sage: P[:100000].list()
[1, 2, 3]
sage: P[1:7:2].list()
[2]
```

TESTS:

Check that the cache is immutable:

```
sage: lazy = lazy_list(iter(Primes()))[:5]
sage: l = lazy.list(); l
[2, 3, 5, 7, 11]
sage: l[0] = -1; l
[-1, 3, 5, 7, 11]
sage: lazy.list()
[2, 3, 5, 7, 11]
```

start_stop_step()

Return the triple (start, stop, step) of reference points of the original lazy list.

EXAMPLES:

```
sage: from sage.misc.lazy_list import lazy_list
sage: p = lazy_list(iter(Primes()))[:2147483647]
sage: p.start_stop_step()
(0, 2147483647, 1)
sage: q = p[100:1042233:12]
sage: q.start_stop_step()
(100, 1042240, 12)
sage: r = q[233::3]
sage: r.start_stop_step()
(2896, 1042252, 36)
sage: 1042241%3 == 233%3
True
```

class sage.misc.lazy_list.**lazy_list_iterator**

Bases: object

Iterator for a lazy list.

INPUT:

- l – a lazy list
- pos – (Default: None) None or a non-negative integer specifying the starting position

next()

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

class sage.misc.lazy_list.**stopped_lazy_list_iterator**

Bases: object

A lazy list iterator which eventually stops.

INPUT:

- l – a lazy list
- pos – (Default: None) None or a non-negative integer specifying the starting position

next()

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

CHAPTER

TEN

CLASS INHERITANCE GRAPHS

`sage.misc.classgraph.`**`class_graph`**`(`*top*, *depth=5*, *name_filter=None*, *classes=None*, *as_graph=True*`)`

Returns the class inheritance graph of a module, class, or object

INPUT:

- `top` – the module, class, or object to start with (e.g. `sage`, `Integer`, `3`)
- `depth` – maximal recursion depth within submodules (default: 5)
- `name_filter` – e.g. 'sage.rings' to only consider classes in `sage.rings`
- `classes` – optional dictionary to be filled in (it is also returned)
- `as_graph` – a boolean (default: True)

OUTPUT:

- An oriented graph, with class names as vertices, and an edge from each class to each of its bases.

EXAMPLES:

We construct the inheritance graph of the classes within a given module:

```
sage: from sage.rings.polynomial.padics import polynomial_padic_capped_relative_dense, polynomia
sage: G = class_graph(sage.rings.polynomial.padics); G
Digraph on 6 vertices
sage: G.vertices()
['Polynomial', 'Polynomial_generic_dense', 'Polynomial_generic_domain', 'Polynomial_padic', 'Pol
sage: G.edges(labels=False)
[('Polynomial_padic', 'Polynomial'), ('Polynomial_padic_capped_relative_dense', 'Polynomial_gene
```

We construct the inheritance graph of a given class:

```
sage: class_graph(Parent).edges(labels=False)
[('CategoryObject', 'SageObject'), ('Parent', 'CategoryObject'), ('SageObject', 'object')]
```

We construct the inheritance graph of the class of an object:

```
sage: class_graph([1,2,3]).edges(labels=False)
[('list', 'object')]
```

Warning: the output of `class_graph` used to be a dictionary mapping each class name to the list of names of its bases. This can be emulated by setting the option `as_graph` to `False`:

```
sage: class_graph(sage.rings.polynomial.padics, depth=2, as_graph=False)
{'Polynomial_padic': ['Polynomial'],
'Polynomial_padic_capped_relative_dense': ['Polynomial_generic_domain', 'Polynomial_padic'],
'Polynomial_padic_flat': ['Polynomial_generic_dense', 'Polynomial_padic']}
```

Note: the `classes` and `as_graph` options are mostly intended for internal recursive use.

Note: `class_graph` does not yet handle nested classes

TESTS:

```
sage: G = class_graph(sage.rings.polynomial.padics, depth=2); G
Digraph on 6 vertices
```

CHAPTER

ELEVEN

SOME TOOLS FOR DEVELOPERS

AUTHORS:

- Nicolas M. Thiery: initial version
- Vincent Delecroix (2012 and 2013): improve import_statements

`sage.misc.dev_tools.`**`find_object_modules`**`(`*obj*`)`

Return a dictionnary whose keys are the names of the modules where `obj` appear and the value at a given module name is the list of names that `obj` have in that module.

It is very unlikely that the output dictionnary has several keys except when `obj` is an instance of a class.

EXAMPLES:

```
sage: from sage.misc.dev_tools import find_object_modules
sage: find_object_modules(RR)
{'sage.rings.real_mpfr': ['RR']}
sage: find_object_modules(ZZ)
{'sage.rings.integer_ring': ['Z', 'ZZ']}
```

Note: It might be a good idea to move this function in `sage.misc.sageinspect`.

`sage.misc.dev_tools.`**`find_objects_from_name`**`(`*name*, *module_name=None*`)`

Return the list of objects from `module_name` whose name is `name`.

If `name` is in the global namespace, the result is a list of length 1 that contains only this object. Otherwise, the function runs through all loaded modules and returns the list of objects whose name matches `name`.

If `module_name` is not `None`, then search only in submodules of `module_name`.

In order to search through more modules you might use the function `load_submodules()`.

EXAMPLES:

```
sage: import sage.misc.dev_tools as dt
sage: dt.find_objects_from_name('FareySymbol')
[<type 'sage.modular.arithgroup.farey_symbol.Farey'>]

sage: import sympy
sage: dt.find_objects_from_name('RR')
[Real Field with 53 bits of precision, RR]
sage: dt.find_objects_from_name('RR', 'sage')
[Real Field with 53 bits of precision]
sage: dt.find_objects_from_name('RR', 'sympy')
[RR]
```

Examples that do not belong to the global namespace but in a loaded module:

```
sage: 'find_objects_from_name' in globals()
False
sage: objs = dt.find_objects_from_name('find_objects_from_name')
sage: len(objs)
1
sage: dt.find_objects_from_name is dt.find_objects_from_name
True
```

sage.misc.dev_tools.**import_statement_string**(*module*, *names*, *lazy*)

Return a (lazy) import statement for `names` from `module`.

INPUT:

- `module` – the name of a module
- `names` – a list of 2-tuples containing names and alias to import
- `lazy` – a boolean: whether to return a lazy import statement

EXAMPLES:

```
sage: import sage.misc.dev_tools as dt
sage: modname = 'sage.misc.dev_tools'
sage: names_and_aliases = [('import_statement_string', 'iss')]
sage: dt.import_statement_string(modname, names_and_aliases, False)
'from sage.misc.dev_tools import import_statement_string as iss'
sage: dt.import_statement_string(modname, names_and_aliases, True)
"lazy_import('sage.misc.dev_tools', 'import_statement_string', 'iss')"
sage: dt.import_statement_string(modname, [('a','b'),('c','c'),('d','e')], False)
'from sage.misc.dev_tools import a as b, c, d as e'
sage: dt.import_statement_string(modname, [(None,None)], False)
'import sage.misc.dev_tools'
```

sage.misc.dev_tools.**import_statements**(**objects*, ***kwds*)

Print import statements for the given objects.

INPUT:

- `*objects` – a sequence of objects or names.
- `lazy` – a boolean (default: `False`) Whether to print a lazy import statement.
- `verbose` – a boolean (default: `True`) Whether to print information in case of ambiguity.
- `answer_as_str` – a boolean (default: `False`) If `True` return a string instead of printing the statement.

EXAMPLES:

```
sage: import_statements(WeylGroup, lazy_attribute)
from sage.combinat.root_system.weyl_group import WeylGroup
from sage.misc.lazy_attribute import lazy_attribute

sage: import_statements(IntegerRing)
from sage.rings.integer_ring import IntegerRing
```

If `lazy` is True, then `lazy_import()` statements are displayed instead:

```
sage: import_statements(WeylGroup, lazy_attribute, lazy=True)
from sage.misc.lazy_import import lazy_import
lazy_import('sage.combinat.root_system.weyl_group', 'WeylGroup')
lazy_import('sage.misc.lazy_attribute', 'lazy_attribute')
```

In principle, the function should also work on object which are instances. In case of ambiguity, one or two warning lines are printed:

```
sage: import_statements(RDF)
from sage.rings.real_double import RDF

sage: import_statements(ZZ)
# ** Warning **: several names for that object: Z, ZZ
from sage.rings.integer_ring import Z

sage: import_statements(euler_phi)
from sage.rings.arith import euler_phi

sage: import_statements(x)
from sage.calculus.predefined import x
```

If you don't like the warning you can disable them with the option `verbose`:

```
sage: import_statements(ZZ, verbose=False)
from sage.rings.integer_ring import Z

sage: import_statements(x, verbose=False)
from sage.calculus.predefined import x
```

If the object has several names, an other way to get the import statement you expect is to use a string instead of the object:

```
sage: import_statements(matrix)
# ** Warning **: several names for that object: Matrix, matrix
from sage.matrix.constructor import Matrix

sage: import_statements('cached_function')
from sage.misc.cachefunc import cached_function
sage: import_statements('Z')
# **Warning**: distinct objects with name 'Z' in:
#   - sage.calculus.predefined
#   - sage.rings.integer_ring
from sage.rings.integer_ring import Z
```

Specifying a string is also useful for objects that are not imported in the Sage interpreter namespace by default. In this case, an object with that name is looked up in all the modules that have been imported in this session:

```
sage: import_statement_string
Traceback (most recent call last):
...
NameError: name 'import_statement_string' is not defined

sage: import_statements("import_statement_string")
from sage.misc.dev_tools import import_statement_string
```

Sometimes objects are imported as an alias (from XXX import YYY as ZZZ) or are affected (XXX = YYY) and the function might dectect it:

```
sage: import_statements('FareySymbol')
from sage.modular.arithgroup.farey_symbol import Farey as FareySymbol

sage: import_statements('sum')
from sage.misc.functional import symbolic_sum as sum

sage: import_statements('power')
```

```
from sage.structure.element import generic_power as power
```

In order to be able to detect functions that belong to a non-loaded module, you might call the helper `load_submodules()` as in the following:

```
sage: import_statements('EnumeratedSetFromIterator')
Traceback (most recent call last):
...
LookupError: no object named 'EnumeratedSetFromIterator'
sage: from sage.misc.dev_tools import load_submodules
sage: load_submodules(sage.sets)
load sage.sets.cartesian_product... succeeded
load sage.sets.set_from_iterator... succeeded
sage: import_statements('EnumeratedSetFromIterator')
from sage.sets.set_from_iterator import EnumeratedSetFromIterator
```

We test different objects which have no appropriate answer:

```
sage: import_statements('my_tailor_is_rich')
Traceback (most recent call last):
...
LookupError: no object named 'my_tailor_is_rich'
sage: import_statements(5)
Traceback (most recent call last):
...
ValueError: no import statement found for '5'.
```

We test that it behaves well with lazy imported objects (trac ticket #14767):

```
sage: import_statements(NN)
from sage.rings.semirings.non_negative_integer_semiring import NN
sage: import_statements('NN')
from sage.rings.semirings.non_negative_integer_semiring import NN
```

Deprecated lazy imports are ignored (see trac ticket #17458):

```
sage: lazy_import('sage.all', 'RR', 'deprecated_RR', namespace=sage.__dict__, deprecation=17458)
sage: import_statements('deprecated_RR')
Traceback (most recent call last):
...
LookupError: object named 'deprecated_RR' is deprecated (see trac ticket 17458)
sage: lazy_import('sage.all', 'RR', namespace=sage.__dict__, deprecation=17458)
sage: import_statements('RR')
from sage.rings.real_mpfr import RR
```

The following were fixed with trac ticket #15351:

```
sage: import_statements('Rationals')
from sage.rings.rational_field import RationalField as Rationals
sage: import_statements(sage.combinat.partition_algebra.SetPartitionsAk)
from sage.combinat.partition_algebra import SetPartitionsAk
sage: import_statements(CIF)
from sage.rings.all import CIF
sage: import_statements(NaN)
from sage.symbolic.constants import NaN
sage: import_statements(pi)
from sage.symbolic.constants import pi
sage: import_statements('SAGE_ENV')
from sage.env import SAGE_ENV
sage: import_statements('graph_decompositions')
```

```
import sage.graphs.graph_decompositions
```

Note: The programmers try to made this function as smart as possible. Nevertheless it is far from being perfect (for example it does not detect deprecated stuff). So, if you use it, double check the answer and report weird behaviors.

`sage.misc.dev_tools.load_submodules`(*module=None*, *exclude_pattern=None*)

Load all submodules of a given modules.

This method is intended to be used by developers and especially the one who uses `import_statements()`. By default it load the sage library and it takes around a minute.

INPUT:

- `module` - an optional module
- `exclude_pattern` - an optional regular expression pattern of module names that have to be excluded.

EXAMPLES:

```
sage: sage.misc.dev_tools.load_submodules(sage.combinat)
load sage.combinat.algebraic_combinatorics... succeeded
...
load sage.combinat.words.suffix_trees... succeeded
```

Calling a second time has no effect (since the function does not import modules already imported):

```
sage: sage.misc.dev_tools.load_submodules(sage.combinat)
```

The second argument allows to exclude a pattern:

```
sage: sage.misc.dev_tools.load_submodules(sage.geometry, "database$|lattice")
load sage.geometry.fan_isomorphism... succeeded
load sage.geometry.hyperplane_arrangement.affine_subspace... succeeded
...
load sage.geometry.riemannian_manifolds.surface3d_generators... succeeded

sage: sage.misc.dev_tools.load_submodules(sage.geometry)
load sage.geometry.polyhedron.lattice_euclidean_group_element... succeeded
load sage.geometry.polyhedron.palp_database... succeeded
load sage.geometry.polyhedron.ppl_lattice_polygon... succeeded
```

`sage.misc.dev_tools.module_names_cmp`(*x*, *y*)

A comparison function for module names.

This function first compares the depth of the modules and then breaks ties by alphabetical order.

See also:

This function is used in `import_statements()`.

TESTS:

```
sage: from sage.misc.dev_tools import module_names_cmp
sage: l = ['a', 'b', 'a.a', 'a.b', 'b.a', 'b.b']
sage: sorted(l, cmp=module_names_cmp)
['a', 'b', 'a.a', 'a.b', 'b.a', 'b.b']
```

`sage.misc.dev_tools.runsnake`(*command*)

Graphical profiling with `runsnake`

INPUT:

- `command` – the command to be run as a string.

EXAMPLES:

```
sage: runsnake("list(SymmetricGroup(3))")        # optional - runsnake
```

`command` is first preparsed (see `preparse()`):

```
sage: runsnake('for x in range(1,4): print x^2') # optional - runsnake
1
4
9
```

`runsnake()` requires the program `runsnake`. Due to non trivial dependencies (python-wxgtk, ...), installing it within the Sage distribution is unpractical. Hence, we recommend installing it with the system wide Python. On Ubuntu 10.10, this can be done with:

```
> sudo apt-get install python-profiler python-wxgtk2.8 python-setuptools
> sudo easy_install RunSnakeRun
```

See the `runsnake` website for instructions for other platforms.

`runsnake()` further assumes that the system wide Python is installed in `/usr/bin/python`.

See also:

- The runsnake website
- `%prun`
- `Profiler`

CHAPTER

TWELVE

FUNCTION MANGLING

This module provides utilities for extracting information about python functions.

AUTHORS:

- Tom Boothby (2009): Original version in Python
- Simon King (2011): Use Cython. Speedup of `fix_to_pos`, cleaning documentation.

class `sage.misc.function_mangling.`**`ArgumentFixer`**

Bases: `object`

This class provides functionality to normalize the arguments passed into a function. While the various ways of calling a function are perfectly equivalent from the perspective of the callee, they don't always look the same for an object watching the caller. For example,

```
sage: def f(x = 10):
...       return min(1,x)
```

the following calls are equivalent,

```
sage: f()
1
sage: f(10)
1
sage: f(x=10)
1
```

but from the perspective of a wrapper, they are different:

```
sage: def wrap(g):
...       def _g(*args,**kwargs):
...           print args, kwargs
...           return g(*args, **kwargs)
...       return _g
sage: h = wrap(f)
sage: t = h()
() {}
sage: t = h(10)
(10,) {}
sage: t = h(x=10)
() {'x': 10}
```

For the purpose of cached functions, it is important not to distinguish between these uses.

INPUTS:

- f – a function

•classmethod – boolean (default False) – True if the function is a classmethod and therefore the first argument is expected to be the class instance. In that case, we ignore the first argument.

EXAMPLES:

```
sage: from sage.misc.function_mangling import ArgumentFixer
sage: def wrap2(g):
...       af = ArgumentFixer(g)
...       def _g(*args, **kwargs):
...           print af.fix_to_pos()
...           return g(*args,**kwargs)
...       return _g
sage: h2 = wrap2(f)
sage: t = h2()
((10,), ())
sage: t = h2(10)
((10,), ())
sage: t = h2(x=10)
((10,), ())

sage: class one:
...      def __init__(self, x = 1):
...         self.x = x
sage: af = ArgumentFixer(one.__init__.__func__, classmethod=True)
sage: af.fix_to_pos(1,2,3,a=31,b=2,n=3)
((1, 2, 3), (('a', 31), ('b', 2), ('n', 3)))
```

defaults_to_pos(*Args*)

f

fix_to_named(**args*, ***kwargs*)

Normalize the arguments with a preference for named arguments.

INPUT:

•any positional and named arguments.

OUTPUT:

We return a tuple

$$(e_1, e_2, ..., e_k), ((n_1, v_1), ..., (n_m, v_m))$$

where $n_1, ..., n_m$ are the names of the arguments and $v_1, ..., v_m$ are the values passed in; and $e_1, ..., e_k$ are the unnamed arguments. We minimize k.

The defaults are extracted from the function and filled into the list `K` of named arguments. The names $n_1, ..., n_t$ are in order of the function definition, where t is the number of named arguments. The remaining names, $n_{t+1}, ..., n_m$ are given in alphabetical order. This is useful to extract the names of arguments, but **does not** maintain equivalence of

```
A,K = self.fix_to_pos(...)
self.f(*A,**dict(K))`
```

and

```
self.f(...)
```

in all cases.

EXAMPLE:

```
sage: from sage.misc.function_mangling import ArgumentFixer
sage: def sum3(a,b,c=3,*args,**kwargs):
...       return a+b+c
sage: AF = ArgumentFixer(sum3)
sage: AF.fix_to_named(1,2,3,4,5,6,f=14,e=16)
((4, 5, 6), (('a', 1), ('b', 2), ('c', 3), ('e', 16), ('f', 14)))
sage: AF.fix_to_named(1,2,f=14)
((), (('a', 1), ('b', 2), ('c', 3), ('f', 14)))
```

fix_to_pos(**args*, ***kwds*)

Normalize the arguments with a preference for positional arguments.

INPUT:

Any positional or named arguments

OUTPUT:

We return a tuple

$$(e_1, e_2, ..., e_k), ((n_1, v_1), ..., (n_m, v_m))$$

where $n_1, ..., n_m$ are the names of the arguments and $v_1, ..., v_m$ are the values passed in; and $e_1, ..., e_k$ are the unnamed arguments. We minimize m.

The commands

```
A,K = self.fix_to_pos(...)
self.f(*A,**dict(K))
```

are equivalent to

```
self.f(...)
```

though defaults are extracted from the function and appended to the tuple `A` of positional arguments. The names $n_1, ..., n_m$ are given in alphabetical order.

EXAMPLE:

```
sage: from sage.misc.function_mangling import ArgumentFixer
sage: def do_something(a,b,c=3,*args,**kwargs):
...       print a,b,c, args, kwargs
sage: AF = ArgumentFixer(do_something)
sage: A,K = AF.fix_to_pos(1,2,3,4,5,6,f=14,e=16); print A,K
(1, 2, 3, 4, 5, 6) (('e', 16), ('f', 14))
sage: do_something(*A,**dict(K))
1 2 3 (4, 5, 6) {'e': 16, 'f': 14}
sage: do_something(1,2,3,4,5,6,f=14,e=16)
1 2 3 (4, 5, 6) {'e': 16, 'f': 14}
```

CHAPTER

THIRTEEN

EXCEPTIONS

This module defines Sage-specific exceptions.

exception sage.misc.exceptions.**OptionalPackageNotFoundError**

Bases: exceptions.RuntimeError

This class defines the exception that should be raised when a function, method, or class cannot detect an optional package that it depends on. When an OptionalPackageNotFoundError is raised, this means one of the following:

- The required optional package is not installed.
- The required optional package is installed, but the relevant interface to that package is unable to detect the package.

EXAMPLES:

```
sage: from sage.misc.exceptions import OptionalPackageNotFoundError
sage: def find_package(fav_package):
...       try:
...           raise OptionalPackageNotFoundError("Unable to detect optional package: %s" % fav_p
...       except OptionalPackageNotFoundError:
...           raise
...
sage: find_package("ham and spam")
Traceback (most recent call last):
...
OptionalPackageNotFoundError: Unable to detect optional package: ham and spam
```

CHAPTER

FOURTEEN

MISCELLANEOUS FUNCTIONS

AUTHORS:

- William Stein
- William Stein (2006-04-26): added workaround for Windows where most users' home directory has a space in it.
- Robert Bradshaw (2007-09-20): Ellipsis range/iterator.

TESTS:

The following test, verifying that trac ticket #16181 has been resolved, needs to stay at the beginning of this file so that its context is not poisoned by other tests:

```
sage: sage.misc.misc.inject_variable('a', 0)
sage: a
0
```

Check the fix from trac ticket #8323:

```
sage: 'name' in globals()
False
sage: 'func' in globals()
False
```

Test deprecation:

```
sage: sage.misc.misc.mul([3,4])
doctest:...: DeprecationWarning:
Importing prod from here is deprecated. If you need to use it, please import it directly from sage.m
See http://trac.sagemath.org/17460 for details.
12
```

class sage.misc.misc.**AttrCallObject**(*name*, *args*, *kwds*)

Bases: object

TESTS:

```
sage: f = attrcall('core', 3); f
*.core(3)
sage: TestSuite(f).run()
```

class sage.misc.misc.**BackslashOperator**

Implements Matlab-style backslash operator for solving systems:

```
A \ b
```

The preparser converts this to multiplications using `BackslashOperator()`.

EXAMPLES:

```
sage: preparse("A \ matrix(QQ,2,1,[1/3,'2/3'])")
"A  * BackslashOperator() * matrix(QQ,Integer(2),Integer(1),[Integer(1)/Integer(3),'2/3'])"
sage: preparse("A \ matrix(QQ,2,1,[1/3,2*3])")
'A  * BackslashOperator() * matrix(QQ,Integer(2),Integer(1),[Integer(1)/Integer(3),Integer(2)*In
sage: preparse("A \ B + C")
'A  * BackslashOperator() * B + C'
sage: preparse("A \ eval('C+D')")
"A  * BackslashOperator() * eval('C+D')"
sage: preparse("A \ x / 5")
'A  * BackslashOperator() * x / Integer(5)'
sage: preparse("A^3 \ b")
'A**Integer(3)  * BackslashOperator() * b'
```

class `sage.misc.misc.`**`GlobalCputime`**(*t*)

Container for CPU times of subprocesses.

AUTHOR:

- Martin Albrecht - (2008-12): initial version

EXAMPLE:

Objects of this type are returned if `subprocesses=True` is passed to `cputime()`:

```
sage: cputime(subprocesses=True) # indirect doctest, output random
0.2347431
```

We can use it to keep track of the CPU time spent in Singular for example:

```
sage: t = cputime(subprocesses=True)
sage: P = PolynomialRing(QQ,7,'x')
sage: I = sage.rings.ideal.Katsura(P)
sage: gb = I.groebner_basis() # calls Singular
sage: cputime(subprocesses=True) - t # output random
0.462987
```

For further processing we can then convert this container to a float:

```
sage: t = cputime(subprocesses=True)
sage: float(t) #output somewhat random
2.1088339999999999
```

See also:

`cputime()`

`sage.misc.misc.`**`alarm`**(*seconds*)

Raise an `AlarmInterrupt` exception in a given number of seconds. This is useful for automatically interrupting long computations and can be trapped using exception handling.

Use `cancel_alarm()` to cancel a previously scheduled alarm.

INPUT:

- `seconds` – positive number, may be floating point

EXAMPLES:

```
sage: alarm(0.5); factor(2^1031-1)
Traceback (most recent call last):
...
```

```
AlarmInterrupt
sage: alarm(0)
Traceback (most recent call last):
...
ValueError: alarm() time must be positive
```

`sage.misc.misc.`**`assert_attribute`**`(`*x*, *attr*, *init=None*`)`

If the object x has the attribute attr, do nothing. If not, set x.attr to init.

`sage.misc.misc.`**`attrcall`**`(`*name*, **args*, ***kwds*`)`

Returns a callable which takes in an object, gets the method named name from that object, and calls it with the specified arguments and keywords.

INPUT:

- `name` - a string of the name of the method you want to call
- `args, kwds` - arguments and keywords to be passed to the method

EXAMPLES:

```
sage: f = attrcall('core', 3); f
*.core(3)
sage: [f(p) for p in Partitions(5)]
[[2], [1, 1], [1, 1], [3, 1, 1], [2], [2], [1, 1]]
```

class `sage.misc.misc.`**`cached_attribute`**`(`*method*, *name=None*`)`

Bases: `object`

Computes attribute value and caches it in the instance.

`sage.misc.misc.`**`call_method`**`(`*obj*, *name*, **args*, ***kwds*`)`

Call the method `name` on `obj`.

This has to exist somewhere in Python!!!

See also:

`operator.methodcaller()` `attrcal()`

EXAMPLES:

```
sage: from sage.misc.misc import call_method
sage: call_method(1, "__add__", 2)
3
```

`sage.misc.misc.`**`cancel_alarm`**`()`

Cancel a previously scheduled alarm (if any) set by `alarm()`.

EXAMPLES:

```
sage: alarm(0.5)
sage: cancel_alarm()
sage: cancel_alarm()   # Calling more than once doesn't matter
sage: sleep(0.6)       # sleep succeeds
```

`sage.misc.misc.`**`cmp_props`**`(`*left*, *right*, *props*`)`

x.__init__(...) initializes x; see help(type(x)) for signature

`sage.misc.misc.`**`coeff_repr`**`(`*c*, *is_latex=False*`)`

x.__init__(...) initializes x; see help(type(x)) for signature

sage.misc.misc.**compose** (*f*, *g*)

Return the composition of one-variable functions: $f \circ g$

See also self_compose() and nest()

INPUT:

- f – a function of one variable
- g – another function of one variable

OUTPUT: A function, such that compose(f,g)(x) = f(g(x))

EXAMPLES:

```
sage: def g(x): return 3*x
sage: def f(x): return x + 1
sage: h1 = compose(f,g)
sage: h2 = compose(g,f)
sage: _ = var ('x')
sage: h1(x)
3*x + 1
sage: h2(x)
3*x + 3
```

:: sage: _ = function('f g') sage: _ = var ('x') sage: compose(f,g)(x) f(g(x))

sage.misc.misc.**cputime** (*t=0*, *subprocesses=False*)

Return the time in CPU seconds since Sage started, or with optional argument `t`, return the time since `t`. This is how much time Sage has spent using the CPU. If `subprocesses=False` this does not count time spent in subprocesses spawned by Sage (e.g., Gap, Singular, etc.). If `subprocesses=True` this function tries to take all subprocesses with a working `cputime()` implementation into account.

The measurement for the main Sage process is done via a call to `resource.getrusage()`, so it avoids the wraparound problems in `time.clock()` on Cygwin.

INPUT:

- `t` - (optional) time in CPU seconds, if `t` is a result from an earlier call with `subprocesses=True`, then `subprocesses=True` is assumed.
- subprocesses – (optional), include subprocesses (default: `False`)

OUTPUT:

- `float` - time in CPU seconds if `subprocesses=False`
- `GlobalCputime` - object which holds CPU times of subprocesses otherwise

EXAMPLES:

```
sage: t = cputime()
sage: F = gp.factor(2^199-1)
sage: cputime(t)          # somewhat random
0.010999000000000092

sage: t = cputime(subprocesses=True)
sage: F = gp.factor(2^199-1)
sage: cputime(t) # somewhat random
0.091999

sage: w = walltime()
sage: F = gp.factor(2^199-1)
```

```
sage: walltime(w)          # somewhat random
0.58425593376159668
```

Note: Even with `subprocesses=True` there is no guarantee that the CPU time is reported correctly because subprocesses can be started and terminated at any given time.

`sage.misc.misc.`**`ellipsis_iter`**`(`**args*, ***kwds*`)`

Same as ellipsis_range, but as an iterator (and may end with an Ellipsis).

See also ellipsis_range.

Use (1,2,...) notation.

EXAMPLES:

```
sage: A = ellipsis_iter(1,2,Ellipsis)
sage: [next(A) for _ in range(10)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: next(A)
11
sage: A = ellipsis_iter(1,3,5,Ellipsis)
sage: [next(A) for _ in range(10)]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
sage: A = ellipsis_iter(1,2,Ellipsis,5,10,Ellipsis)
sage: [next(A) for _ in range(10)]
[1, 2, 3, 4, 5, 10, 11, 12, 13, 14]
```

TESTS:

These were carefully chosen tests, only to be changed if the semantics of ellipsis ranges change. In other words, if they don't pass it's probably a bug in the implementation, not in the doctest.

```
sage: list(1,..,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: list(1,3,..,10)
[1, 3, 5, 7, 9]
sage: list(1,..,10,..,20)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
sage: list(1,3,..,10,..,20)
[1, 3, 5, 7, 9, 10, 12, 14, 16, 18, 20]
sage: list(1,3,..,10,10,..,20)
[1, 3, 5, 7, 9, 10, 12, 14, 16, 18, 20]
sage: list(0,2,..,10,10,..,20,20,..,25)
[0, 2, 4, 6, 8, 10, 10, 12, 14, 16, 18, 20, 20, 22, 24]
sage: list(10,..,1)
[]
sage: list(10,11,..,1)
[]
sage: list(10,9,..,1)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
sage: list(100,..,10,..,20)
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
sage: list(0,..,10,..,-20)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: list(100,..,10,..,-20)
[]
sage: list(100,102,..,10,..,20)
[10, 12, 14, 16, 18, 20]
```

sage.misc.misc.**ellipsis_range**(**args*, ***kwds*)

Return arithmetic sequence determined by the numeric arguments and ellipsis. Best illustrated by examples.

Use [1,2,..,n] notation.

EXAMPLES:

```
sage: ellipsis_range(1,Ellipsis,11,100)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 100]
sage: ellipsis_range(0,2,Ellipsis,10,Ellipsis,20)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
sage: ellipsis_range(0,2,Ellipsis,11,Ellipsis,20)
[0, 2, 4, 6, 8, 10, 11, 13, 15, 17, 19]
sage: ellipsis_range(0,2,Ellipsis,11,Ellipsis,20, step=3)
[0, 2, 5, 8, 11, 14, 17, 20]
sage: ellipsis_range(10,Ellipsis,0)
[]
```

TESTS: These were carefully chosen tests, only to be changed if the semantics of ellipsis ranges change. In other words, if they don't pass it's probably a bug in the implementation, not in the doctest.

Note 10 only appears once (though it is in both ranges).

```
sage: ellipsis_range(0,Ellipsis,10,Ellipsis,20,step=2)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Sometimes one or more ranges is empty.

```
sage: ellipsis_range(100,Ellipsis,10,Ellipsis,20,step=2)
[10, 12, 14, 16, 18, 20]
sage: ellipsis_range(0,Ellipsis,10,Ellipsis,-20,step=2)
[0, 2, 4, 6, 8, 10]
sage: ellipsis_range(100,Ellipsis,10,Ellipsis,-20,step=2)
[]
```

We always start on the leftmost point of the range.

```
sage: ellipsis_range(0,Ellipsis,10,Ellipsis,20,step=3)
[0, 3, 6, 9, 10, 13, 16, 19]
sage: ellipsis_range(100,Ellipsis,10,Ellipsis,20,step=3)
[10, 13, 16, 19]
sage: ellipsis_range(0,Ellipsis,10,Ellipsis,-20,step=3)
[0, 3, 6, 9]
sage: ellipsis_range(100,Ellipsis,10,Ellipsis,-20,step=3)
[]
sage: ellipsis_range(0,1,Ellipsis,-10)
[]
sage: ellipsis_range(0,1,Ellipsis,-10,step=1)
[0]
sage: ellipsis_range(100,0,1,Ellipsis,-10)
[100]
```

Note the duplicate 5 in the output.

```
sage: ellipsis_range(0,Ellipsis,5,5,Ellipsis,10)
[0, 1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10]
```

Examples in which the step determines the parent of the elements:

```
sage: [1..3, step=0.5]
[1.00000000000000, 1.50000000000000, 2.00000000000000, 2.50000000000000, 3.00000000000000]
sage: v = [1..5, step=1/1]; v
```

```
[1, 2, 3, 4, 5]
sage: parent(v[2])
Rational Field
```

sage.misc.misc.**embedded**()

Return True if this copy of Sage is running embedded in the Sage notebook.

EXAMPLES:

```
sage: sage.misc.misc.embedded()    # output True if in the notebook
False
```

sage.misc.misc.**exists**(*S*, *P*)

If S contains an element x such that P(x) is True, this function returns True and the element x. Otherwise it returns False and None.

Note that this function is NOT suitable to be used in an if-statement or in any place where a boolean expression is expected. For those situations, use the Python built-in

any(P(x) for x in S)

INPUT:

- `S` - object (that supports enumeration)
- `P` - function that returns True or False

OUTPUT:

- `bool` - whether or not P is True for some element x of S
- `object` - x

EXAMPLES: lambda functions are very useful when using the exists function:

```
sage: exists([1,2,5], lambda x : x > 7)
(False, None)
sage: exists([1,2,5], lambda x : x > 3)
(True, 5)
```

The following example is similar to one in the MAGMA handbook. We check whether certain integers are a sum of two (small) cubes:

```
sage: cubes = [t**3 for t in range(-10,11)]
sage: exists([(x,y) for x in cubes for y in cubes], lambda v : v[0]+v[1] == 218)
(True, (-125, 343))
sage: exists([(x,y) for x in cubes for y in cubes], lambda v : v[0]+v[1] == 219)
(False, None)
```

sage.misc.misc.**forall**(*S*, *P*)

If P(x) is true every x in S, return True and None. If there is some element x in S such that P is not True, return False and x.

Note that this function is NOT suitable to be used in an if-statement or in any place where a boolean expression is expected. For those situations, use the Python built-in

all(P(x) for x in S)

INPUT:

- `S` - object (that supports enumeration)
- `P` - function that returns True or False

OUTPUT:

- `bool` - whether or not P is True for all elements of S
- `object` - x

EXAMPLES: lambda functions are very useful when using the forall function. As a toy example we test whether certain integers are greater than 3.

```
sage: forall([1,2,5], lambda x : x > 3)
(False, 1)
sage: forall([1,2,5], lambda x : x > 0)
(True, None)
```

Next we ask whether every positive integer less than 100 is a product of at most 2 prime factors:

```
sage: forall(range(1,100),  lambda n : len(factor(n)) <= 2)
(False, 30)
```

The answer is no, and 30 is a counterexample. However, every positive integer 100 is a product of at most 3 primes.

```
sage: forall(range(1,100),  lambda n : len(factor(n)) <= 3)
(True, None)
```

sage.misc.misc.**generic_cmp**(*x*, *y*)

Compare x and y and return -1, 0, or 1.

This is similar to x.__cmp__(y), but works even in some cases when a .__cmp__ method isn't defined.

sage.misc.misc.**get_main_globals**()

Return the main global namespace.

EXAMPLES:

```
sage: from sage.misc.misc import get_main_globals
sage: G = get_main_globals()
sage: bla = 1
sage: G['bla']
1
sage: bla = 2
sage: G['bla']
2
sage: G['ble'] = 5
sage: ble
5
```

This is analogous to `globals()`, except that it can be called from any function, even if it is in a Python module:

```
sage: def f():
....:     G = get_main_globals()
....:     assert G['bli'] == 14
....:     G['blo'] = 42
sage: bli = 14
sage: f()
sage: blo
42
```

ALGORITHM:

The main global namespace is discovered by going up the frame stack until the frame for the `__main__` module is found. Should this frame not be found (this should not occur in normal operation), an exception "ValueError:

call stack is not deep enough" will be raised by `_getframe`.

See `inject_variable_test()` for a real test that this works within deeply nested calls in a function defined in a Python module.

`sage.misc.misc.`**`get_verbose`**`()`

Return the global Sage verbosity level.

INPUT: int level: an integer between 0 and 2, inclusive.

OUTPUT: changes the state of the verbosity flag.

EXAMPLES:

```
sage: get_verbose()
0
sage: set_verbose(2)
sage: get_verbose()
2
sage: set_verbose(0)
```

`sage.misc.misc.`**`get_verbose_files`**`()`

`sage.misc.misc.`**`getitem`**`(v, n)`

Variant of getitem that coerces to an int if a TypeError is raised.

(This is not needed anymore - classes should define an __index__ method.)

Thus, e.g., `getitem(v,n)` will work even if v is a Python list and n is a Sage integer.

EXAMPLES:

```
sage: v = [1,2,3]
```

The following used to fail in Sage <= 1.3.7. Now it works fine:

```
sage: v[ZZ(1)]
2
```

This always worked.

```
sage: getitem(v, ZZ(1))
2
```

`sage.misc.misc.`**`inject_variable`**`(name, value)`

Inject a variable into the main global namespace.

INPUT:

- `name` – a string
- `value` – anything

EXAMPLES:

```
sage: from sage.misc.misc import inject_variable
sage: inject_variable("a", 314)
sage: a
314
```

A warning is issued the first time an existing value is overwritten:

```
sage: inject_variable("a", 271)
doctest:...: RuntimeWarning: redefining global value `a`
sage: a
271
```

```
sage: inject_variable("a", 272)
sage: a
272
```

That's because warn seem to not reissue twice the same warning:

> sage: from warnings import warn sage: warn("blah") doctest:...: UserWarning: blah sage: warn("blah")

Use with care!

`sage.misc.misc.`**`inject_variable_test`**(*name*, *value*, *depth*)

A function for testing deep calls to inject_variable

TESTS:

```
sage: from sage.misc.misc import inject_variable_test
sage: inject_variable_test("a0", 314, 0)
sage: a0
314
sage: inject_variable_test("a1", 314, 1)
sage: a1
314
sage: inject_variable_test("a2", 314, 2)
sage: a2
314
sage: inject_variable_test("a2", 271, 2)
doctest:...: RuntimeWarning: redefining global value `a2`
sage: a2
271
```

`sage.misc.misc.`**`is_in_string`**(*line*, *pos*)

Returns True if the character at position pos in line occurs within a string.

EXAMPLES:

```
sage: from sage.misc.misc import is_in_string
sage: line = 'test(\'#\')'
sage: is_in_string(line, line.rfind('#'))
True
sage: is_in_string(line, line.rfind(')'))
False
```

`sage.misc.misc.`**`is_iterator`**(*it*)

Tests if it is an iterator.

The mantra `if hasattr(it, 'next')` was used to tests if `it` is an iterator. This is not quite correct since `it` could have a `next` methods with a different semantic.

EXAMPLES:

```
sage: it = iter([1,2,3])
sage: is_iterator(it)
True

sage: class wrong():
...       def __init__(self): self.n = 5
...       def next(self):
...           self.n -= 1
...           if self.n == 0: raise StopIteration
...           return self.n
```

```
sage: x = wrong()
sage: is_iterator(x)
False
sage: list(x)
Traceback (most recent call last):
...
TypeError: iteration over non-sequence

sage: class good(wrong):
...       def __iter__(self): return self
sage: x = good()
sage: is_iterator(x)
True
sage: list(x)
[4, 3, 2, 1]

sage: P = Partitions(3)
sage: is_iterator(P)
False
sage: is_iterator(iter(P))
True
```

class `sage.misc.misc.`**`lazy_prop`**(*calculate_function*)

Bases: `object`

`sage.misc.misc.`**`nest`**(*f, n, x*)

Return $f(f(...f(x)...))$, where the composition occurs n times.

See also `compose()` and `self_compose()`

INPUT:

- f – a function of one variable
- n – a nonnegative integer
- x – any input for f

OUTPUT: $f(f(...f(x)...))$, where the composition occurs n times

EXAMPLES:

```
sage: def f(x): return x^2 + 1
sage: x = var('x')
sage: nest(f, 3, x)
((x^2 + 1)^2 + 1)^2 + 1

sage: _ = function('f')
sage: _ = var('x')
sage: nest(f, 10, x)
f(f(f(f(f(f(f(f(f(f(x))))))))))

sage: _ = function('f')
sage: _ = var('x')
sage: nest(f, 0, x)
x
```

`sage.misc.misc.`**`newton_method_sizes`**(*N*)

Returns a sequence of integers $1 = a_1 \le a_2 \le \cdots \le a_n = N$ such that $a_j = \lceil a_{j+1}/2 \rceil$ for all j.

This is useful for Newton-style algorithms that double the precision at each stage. For example if you start at precision 1 and want an answer to precision 17, then it's better to use the intermediate stages 1, 2, 3, 5, 9, 17 than to use 1, 2, 4, 8, 16, 17.

INPUT:

- `N` - positive integer

EXAMPLES:

```
sage: newton_method_sizes(17)
[1, 2, 3, 5, 9, 17]
sage: newton_method_sizes(16)
[1, 2, 4, 8, 16]
sage: newton_method_sizes(1)
[1]
```

AUTHORS:

- David Harvey (2006-09-09)

`sage.misc.misc.`**`pad_zeros`**(*s*, *size=3*)

EXAMPLES:

```
sage: pad_zeros(100)
'100'
sage: pad_zeros(10)
'010'
sage: pad_zeros(10, 5)
'00010'
sage: pad_zeros(389, 5)
'00389'
sage: pad_zeros(389, 10)
'0000000389'
```

`sage.misc.misc.`**`powerset`**(*X*)

Iterator over the *list* of all subsets of the iterable X, in no particular order. Each list appears exactly once, up to order.

INPUT:

- `X` - an iterable

OUTPUT: iterator of lists

EXAMPLES:

```
sage: list(powerset([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
sage: [z for z in powerset([0,[1,2]])]
[[], [0], [[1, 2]], [0, [1, 2]]]
```

Iterating over the power set of an infinite set is also allowed:

```
sage: i = 0
sage: for x in powerset(ZZ):
...     if i > 10:
...         break
...     else:
...         i += 1
...     print x,
[] [0] [1] [0, 1] [-1] [0, -1] [1, -1] [0, 1, -1] [2] [0, 2] [1, 2]
```

You may also use subsets as an alias for powerset:

```
sage: subsets([1,2,3])
<generator object powerset at 0x...>
sage: list(subsets([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

The reason we return lists instead of sets is that the elements of
sets must be hashable and many structures on which one wants the
powerset consist of non-hashable objects.
```

AUTHORS:

- William Stein
- Nils Bruin (2006-12-19): rewrite to work for not-necessarily finite objects X.

`sage.misc.misc.`**`prop`**(*f*)

x.__init__(...) initializes x; see help(type(x)) for signature

`sage.misc.misc.`**`random_sublist`**(*X*, *s*)

Return a pseudo-random sublist of the list X where the probability of including a particular element is s.

INPUT:

- `X` - list
- `s` - floating point number between 0 and 1

OUTPUT: list

EXAMPLES:

```
sage: S = [1,7,3,4,18]
sage: random_sublist(S, 0.5)
[1, 3, 4]
sage: random_sublist(S, 0.5)
[1, 3]
```

`sage.misc.misc.`**`repr_lincomb`**(*terms*, *coeffs=None*, *is_latex=False*, *scalar_mult='*'*, *strip_one=False*, *repr_monomial=None*, *latex_scalar_mult=None*)

Compute a string representation of a linear combination of some formal symbols.

INPUT:

- `terms` – list of terms, as pairs (support, coefficient)
- `is_latex` – whether to produce latex (default: `False`)
- `scalar_mult` – string representing the multiplication (default:`'*'`)
- `latex_scalar_mult` – latex string representing the multiplication (default: `''` if `scalar_mult` is `'*'`; otherwise `scalar_mult`)
- `coeffs` – for backward compatibility

OUTPUT:

- `str` - a string

EXAMPLES:

```
sage: repr_lincomb([('a',1), ('b',-2), ('c',3)])
'a - 2*b + 3*c'
sage: repr_lincomb([('a',0), ('b',-2), ('c',3)])
'-2*b + 3*c'
sage: repr_lincomb([('a',0), ('b',2), ('c',3)])
```

```
'2*b + 3*c'
sage: repr_lincomb([('a',1), ('b',0), ('c',3)])
'a + 3*c'
sage: repr_lincomb([('a',-1), ('b','2+3*x'), ('c',3)])
'-a + (2+3*x)*b + 3*c'
sage: repr_lincomb([('a', '1+x^2'), ('b', '2+3*x'), ('c', 3)])
'(1+x^2)*a + (2+3*x)*b + 3*c'
sage: repr_lincomb([('a', '1+x^2'), ('b', '-2+3*x'), ('c', 3)])
'(1+x^2)*a + (-2+3*x)*b + 3*c'
sage: repr_lincomb([('a', 1), ('b', -2), ('c', -3)])
'a - 2*b - 3*c'
sage: t = PolynomialRing(RationalField(),'t').gen()
sage: repr_lincomb([('a', -t), ('s', t - 2), ('', t^2 + 2)])
'-t*a + (t-2)*s + (t^2+2)'
```

Examples for `scalar_mult`:

```
sage: repr_lincomb([('a',1), ('b',2), ('c',3)], scalar_mult='*')
'a + 2*b + 3*c'
sage: repr_lincomb([('a',2), ('b',0), ('c',-3)], scalar_mult='**')
'2**a - 3**c'
sage: repr_lincomb([('a',-1), ('b',2), ('c',3)], scalar_mult='**')
'-a + 2**b + 3**c'
```

Examples for `scalar_mult` and `is_latex`:

```
sage: repr_lincomb([('a',-1), ('b',2), ('c',3)], is_latex=True)
'-a + 2b + 3c'
sage: repr_lincomb([('a',-1), ('b',-1), ('c',3)], is_latex=True, scalar_mult='*')
'-a - b + 3c'
sage: repr_lincomb([('a',-1), ('b',2), ('c',-3)], is_latex=True, scalar_mult='**')
'-a + 2**b - 3**c'
sage: repr_lincomb([('a',-2), ('b',-1), ('c',-3)], is_latex=True, latex_scalar_mult='*')
'-2*a - b - 3*c'
```

Examples for `strip_one`:

```
sage: repr_lincomb([ ('a',1), (1,-2), ('3',3) ])
'a - 2*1 + 3*3'
sage: repr_lincomb([ ('a',-1), (1,1), ('3',3) ])
'-a + 1 + 3*3'
sage: repr_lincomb([ ('a',1), (1,-2), ('3',3) ], strip_one = True)
'a - 2 + 3*3'
sage: repr_lincomb([ ('a',-1), (1,1), ('3',3) ], strip_one = True)
'-a + 1 + 3*3'
sage: repr_lincomb([ ('a',1), (1,-1), ('3',3) ], strip_one = True)
'a - 1 + 3*3'
```

Examples for `repr_monomial`:

```
sage: repr_lincomb([('a',1), ('b',2), ('c',3)], repr_monomial = lambda s: s+"1")
'a1 + 2*b1 + 3*c1'
```

TESTS:

For backward compatibility (will be deprecated):

```
sage: repr_lincomb(['a','b','c'], [1,2,3])
doctest:...: DeprecationWarning: calling `repr_lincomb(monoms, coeffs)` is deprecated; please sp
See http://trac.sagemath.org/12484 for details.
'a + 2*b + 3*c'
```

`sage.misc.misc.sage_makedirs(dir)`

Python version of `mkdir -p`: try to create a directory, and also create all intermediate directories as necessary. Succeed silently if the directory already exists (unlike `os.makedirs()`). Raise other errors (like permission errors) normally.

EXAMPLES:

```
sage: from sage.misc.misc import sage_makedirs
sage: sage_makedirs(DOT_SAGE) # no output
```

The following fails because we are trying to create a directory in place of an ordinary file (the main Sage executable):

```
sage: sage_executable = os.path.join(SAGE_ROOT, 'sage')
sage: sage_makedirs(sage_executable)
Traceback (most recent call last):
...
OSError: ...
```

`sage.misc.misc.self_compose(f, n)`

Return the function f composed with itself n times.

See `nest()` if you want $f(f(...(f(x))...))$ for known x.

INPUT:

- f – a function of one variable
- n – a nonnegative integer

OUTPUT: A function, the result of composing f with itself n times

EXAMPLES:

```
sage: def f(x): return x^2 + 1
sage: g = self_compose(f, 3)
sage: x = var('x')
sage: g(x)
((x^2 + 1)^2 + 1)^2 + 1

sage: def f(x): return x + 1
sage: g = self_compose(f, 10000)
sage: g(0)
10000

sage: x = var('x')
sage: self_compose(sin, 0)(x)
x
```

`sage.misc.misc.set_verbose(level, files='all')`

Set the global Sage verbosity level.

INPUT:

- `level` - an integer between 0 and 2, inclusive.
- **`files` (default: 'all'): list of files to make verbose, or** 'all' to make ALL files verbose (the default).

OUTPUT: changes the state of the verbosity flag and possibly appends to the list of files that are verbose.

EXAMPLES:

```
sage: set_verbose(2)
sage: verbose("This is Sage.", level=1)  # not tested
VERBOSE1 (?): This is Sage.
sage: verbose("This is Sage.", level=2)  # not tested
VERBOSE2 (?): This is Sage.
sage: verbose("This is Sage.", level=3)  # not tested
[no output]
sage: set_verbose(0)
```

sage.misc.misc.**set_verbose_files**(*file_name*)

sage.misc.misc.**sourcefile**(*object*)

Work out which source or compiled file an object was defined in.

sage.misc.misc.**srange**(*start*, *end=None*, *step=1*, *universe=None*, *check=True*, *include_endpoint=False*, *endpoint_tolerance=1e-05*)

Return list of numbers `a, a+step, ..., a+k*step`, where `a+k*step < b` and `a+(k+1)*step >= b` over exact rings, and makes a best attempt for inexact rings (see note below).

This provides one way to iterate over Sage integers as opposed to Python int's. It also allows you to specify step sizes for such an iteration. Note, however, that what is returned is a full list of Integers and not an iterator. It is potentially much slower than the Python range function, depending on the application. The function xsrange() provides an iterator with similar functionality which would usually be more efficient than using srange().

INPUT:

- `a` - number
- `b` - number (default: None)
- `step` - number (default: 1)
- `universe` - Parent or type where all the elements should live (default: deduce from inputs)
- `check` - make sure a, b, and step all lie in the same universe
- `include_endpoint` - whether or not to include the endpoint (default: False)
- `endpoint_tolerance` - used to determine whether or not the endpoint is hit for inexact rings (default 1e-5)

OUTPUT:

- list

If b is None, then b is set equal to a and a is set equal to the 0 in the parent of b.

Unlike range, a and b can be any type of numbers, and the resulting list involves numbers of that type.

Note: The list elements are computed via repeated addition rather than multiplication, which may produce slightly different results with inexact rings. For example:

```
sage: sum([1.1] * 10) == 1.1 * 10
False
```

Also, the question of whether the endpoint is hit exactly for a given `a + k*step` is fuzzy for an inexact ring. If `a + k*step = b` for some k within `endpoint_tolerance` of being integral, it is considered an exact hit, thus avoiding spurious values falling just below the endpoint.

Note: This function is called `srange` to distinguish it from the built-in Python `range` command. The s at the beginning of the name stands for "Sage".

EXAMPLES:

```
sage: v = srange(5); v
[0, 1, 2, 3, 4]
sage: type(v[2])
<type 'sage.rings.integer.Integer'>
sage: srange(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: srange(10, 1, -1)
[10, 9, 8, 7, 6, 5, 4, 3, 2]
sage: srange(10,1,-1, include_endpoint=True)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
sage: srange(1, 10, universe=RDF)
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]

sage: srange(1, 10, 1/2)
[1, 3/2, 2, 5/2, 3, 7/2, 4, 9/2, 5, 11/2, 6, 13/2, 7, 15/2, 8, 17/2, 9, 19/2]
sage: srange(1, 5, 0.5)
[1.00000000000000, 1.50000000000000, 2.00000000000000, 2.50000000000000, 3.00000000000000, 3.500
sage: srange(0, 1, 0.4)
[0.000000000000000, 0.400000000000000, 0.800000000000000]
sage: srange(1.0, 5.0, include_endpoint=True)
[1.00000000000000, 2.00000000000000, 3.00000000000000, 4.00000000000000, 5.00000000000000]
sage: srange(1.0, 1.1)
[1.00000000000000]
sage: srange(1.0, 1.0)
[]
sage: V = VectorSpace(QQ, 2)
sage: srange(V([0,0]), V([5,5]), step=V([2,2]))
[(0, 0), (2, 2), (4, 4)]
```

Including the endpoint:

```
sage: srange(0, 10, step=2, include_endpoint=True)
[0, 2, 4, 6, 8, 10]
sage: srange(0, 10, step=3, include_endpoint=True)
[0, 3, 6, 9]
```

Try some inexact rings:

```
sage: srange(0.5, 1.1, 0.1, universe=RDF, include_endpoint=False)
[0.5, 0.6, 0.7, 0.7999999999999999, 0.8999999999999999, 0.9999999999999999]
sage: srange(0.5, 1, 0.1, universe=RDF, include_endpoint=False)
[0.5, 0.6, 0.7, 0.7999999999999999, 0.8999999999999999]
sage: srange(0.5, 0.9, 0.1, universe=RDF, include_endpoint=False)
[0.5, 0.6, 0.7, 0.7999999999999999]
sage: srange(0, 1.1, 0.1, universe=RDF, include_endpoint=True)
[0.0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6, 0.7, 0.7999999999999999, 0.8999999999999999,
sage: srange(0, 0.2, 0.1, universe=RDF, include_endpoint=True)
[0.0, 0.1, 0.2]
sage: srange(0, 0.3, 0.1, universe=RDF, include_endpoint=True)
[0.0, 0.1, 0.2, 0.3]
```

TESTS:

These are doctests from trac ticket #6409:

```
sage: srange(1,0,include_endpoint=True)
[]
sage: srange(1,QQ(0),include_endpoint=True)
[]
```

```
sage: srange(3,0,-1,include_endpoint=True)
[3, 2, 1, 0]
sage: srange(1,1,0) # trac ticket #11753
Traceback (most recent call last):
...
ValueError: srange() step argument must not be zero
```

sage.misc.misc.**strunc**(*s*, *n=60*)
Truncate at first space after position n, adding '...' if nontrivial truncation.

sage.misc.misc.**subsets**(*X*)
Iterator over the *list* of all subsets of the iterable X, in no particular order. Each list appears exactly once, up to order.

INPUT:

- X - an iterable

OUTPUT: iterator of lists

EXAMPLES:

```
sage: list(powerset([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
sage: [z for z in powerset([0,[1,2]])]
[[], [0], [[1, 2]], [0, [1, 2]]]
```

Iterating over the power set of an infinite set is also allowed:

```
sage: i = 0
sage: for x in powerset(ZZ):
...     if i > 10:
...        break
...     else:
...        i += 1
...     print x,
[] [0] [1] [0, 1] [-1] [0, -1] [1, -1] [0, 1, -1] [2] [0, 2] [1, 2]
```

You may also use subsets as an alias for powerset:

```
sage: subsets([1,2,3])
<generator object powerset at 0x...>
sage: list(subsets([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

The reason we return lists instead of sets is that the elements of
sets must be hashable and many structures on which one wants the
powerset consist of non-hashable objects.
```

AUTHORS:

- William Stein
- Nils Bruin (2006-12-19): rewrite to work for not-necessarily finite objects X.

sage.misc.misc.**sxrange**(*start*, *end=None*, *step=1*, *universe=None*, *check=True*, *include_endpoint=False*, *endpoint_tolerance=1e-05*)
Return an iterator over numbers `a, a+step, ..., a+k*step`, where `a+k*step < b` and `a+(k+1)*step > b`.

INPUT: universe – Parent or type where all the elements should live (default: deduce from inputs) check – make sure a, b, and step all lie in the same universe include_endpoint – whether or not to include the endpoint (default: False) endpoint_tolerance – used to determine whether or not the endpoint is hit for inexact rings (default 1e-5)

- `a` - number
- `b` - number
- `step` - number (default: 1)

OUTPUT: iterator

Unlike range, a and b can be any type of numbers, and the resulting iterator involves numbers of that type.

See also:

`srange()`

Note: This function is called `xsrange` to distinguish it from the builtin Python `xrange` command.

EXAMPLES:

```
sage: list(xsrange(1,10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: Q = RationalField()
sage: list(xsrange(1, 10, Q('1/2')))
[1, 3/2, 2, 5/2, 3, 7/2, 4, 9/2, 5, 11/2, 6, 13/2, 7, 15/2, 8, 17/2, 9, 19/2]
sage: list(xsrange(1, 5, 0.5))
[1.00000000000000, 1.50000000000000, 2.00000000000000, 2.50000000000000, 3.00000000000000, 3.500
sage: list(xsrange(0, 1, 0.4))
[0.000000000000000, 0.400000000000000, 0.800000000000000]
```

Negative ranges are also allowed:

```
sage: list(xrange(4,1,-1))
[4, 3, 2]
sage: list(sxrange(4,1,-1))
[4, 3, 2]
sage: list(sxrange(4,1,-1/2))
[4, 7/2, 3, 5/2, 2, 3/2]
```

TESTS:

These are doctests from trac ticket #6409:

```
sage: list(xsrange(1,QQ(0),include_endpoint=True))
[]
sage: list(xsrange(1,QQ(0),-1,include_endpoint=True))
[1, 0]
sage: xsrange(1,1,0) # trac ticket #11753
Traceback (most recent call last):
...
ValueError: xsrange() step argument must not be zero
```

sage.misc.misc.**to_gmp_hex**(*n*)
x.__init__(...) initializes x; see help(type(x)) for signature

sage.misc.misc.**todo**(*mesg=''*)
x.__init__(...) initializes x; see help(type(x)) for signature

sage.misc.misc.**typecheck**(*x, C, var='x'*)
Check that x is of instance C. If not raise a TypeError with an error message.

sage.misc.misc.**union**(*x*, *y=None*)

Return the union of x and y, as a list. The resulting list need not be sorted and can change from call to call.

INPUT:

- x - iterable
- y - iterable (may optionally omitted)

OUTPUT: list

EXAMPLES:

```
sage: answer = union([1,2,3,4], [5,6]); answer
[1, 2, 3, 4, 5, 6]
sage: union([1,2,3,4,5,6], [5,6]) == answer
True
sage: union((1,2,3,4,5,6), [5,6]) == answer
True
sage: union((1,2,3,4,5,6), set([5,6])) == answer
True
```

sage.misc.misc.**uniq**(*x*)

Return the sublist of all elements in the list x that is sorted and is such that the entries in the sublist are unique.

EXAMPLES:

```
sage: v = uniq([1,1,8,-5,3,-5,'a','x','a'])
sage: v              # potentially random ordering of output
['a', 'x', -5, 1, 3, 8]
sage: set(v) == set(['a', 'x', -5, 1, 3, 8])
True
```

sage.misc.misc.**unset_verbose_files**(*file_name*)

sage.misc.misc.**verbose**(*mesg='‘*, *t=0*, *level=1*, *caller_name=None*)

Print a message if the current verbosity is at least level.

INPUT:

- mesg - str, a message to print
- t - int, optional, if included, will also print cputime(t), - which is the time since time t. Thus t should have been obtained with t=cputime()
- level - int, (default: 1) the verbosity level of what we are printing
- caller_name - string (default: None), the name of the calling function; in most cases Python can deduce this, so it need not be provided.

OUTPUT: possibly prints a message to stdout; also returns cputime()

EXAMPLE:

```
sage: set_verbose(1)
sage: t = cputime()
sage: t = verbose("This is Sage.", t, level=1, caller_name="william")       # not tested
VERBOSE1 (william): This is Sage. (time = 0.0)
sage: set_verbose(0)
```

sage.misc.misc.**walltime**(*t=0*)

Return the wall time in second, or with optional argument t, return the wall time since time t. “Wall time” means the time on a wall clock, i.e., the actual time.

INPUT:

- `t` - (optional) float, time in CPU seconds

OUTPUT:

- `float` - time in seconds

EXAMPLES:

```
sage: w = walltime()
sage: F = factor(2^199-1)
sage: walltime(w)    # somewhat random
0.8823847770690918
```

`sage.misc.misc.`**`word_wrap`**`(`*s*, *ncols=85*`)`

x.__init__(...) initializes x; see help(type(x)) for signature

`sage.misc.misc.`**`xsrange`**`(`*start*, *end=None*, *step=1*, *universe=None*, *check=True*, *include_endpoint=False*, *endpoint_tolerance=1e-05*`)`

Return an iterator over numbers `a, a+step, ..., a+k*step`, where `a+k*step < b` and `a+(k+1)*step > b`.

INPUT: universe – Parent or type where all the elements should live (default: deduce from inputs) check – make sure a, b, and step all lie in the same universe include_endpoint – whether or not to include the endpoint (default: False) endpoint_tolerance – used to determine whether or not the endpoint is hit for inexact rings (default 1e-5)

- `a` - number
- `b` - number
- `step` - number (default: 1)

OUTPUT: iterator

Unlike range, a and b can be any type of numbers, and the resulting iterator involves numbers of that type.

See also:

`srange()`

Note: This function is called `xsrange` to distinguish it from the builtin Python `xrange` command.

EXAMPLES:

```
sage: list(xsrange(1,10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: Q = RationalField()
sage: list(xsrange(1, 10, Q('1/2')))
[1, 3/2, 2, 5/2, 3, 7/2, 4, 9/2, 5, 11/2, 6, 13/2, 7, 15/2, 8, 17/2, 9, 19/2]
sage: list(xsrange(1, 5, 0.5))
[1.00000000000000, 1.50000000000000, 2.00000000000000, 2.50000000000000, 3.00000000000000, 3.500
sage: list(xsrange(0, 1, 0.4))
[0.000000000000000, 0.400000000000000, 0.800000000000000]
```

Negative ranges are also allowed:

```
sage: list(xrange(4,1,-1))
[4, 3, 2]
sage: list(sxrange(4,1,-1))
[4, 3, 2]
sage: list(sxrange(4,1,-1/2))
[4, 7/2, 3, 5/2, 2, 3/2]
```

TESTS:

These are doctests from trac ticket #6409:

```
sage: list(xsrange(1,QQ(0),include_endpoint=True))
[]
sage: list(xsrange(1,QQ(0),-1,include_endpoint=True))
[1, 0]
sage: xsrange(1,1,0) # trac ticket #11753
Traceback (most recent call last):
...
ValueError: xsrange() step argument must not be zero
```

CHAPTER

FIFTEEN

TEMPORARY FILE HANDLING

AUTHORS:

- Volker Braun, Jeroen Demeyer (2012-10-18): move these functions here from sage/misc/misc.py and make them secure, see trac ticket #13579.
- Jeroen Demeyer (2013-03-17): add class:$atomic_write$, see trac ticket #14292.

class sage.misc.temporary_file.**atomic_write**(*target_filename*, *append=False*, *mode=438*)

Write to a given file using a temporary file and then rename it to the target file. This renaming should be atomic on modern operating systems. Therefore, this class can be used to avoid race conditions when a file might be read while it is being written. It also avoids having partially written files due to exceptions or crashes.

This is to be used in a with statement, where a temporary file is created when entering the with and is moved in place of the target file when exiting the with (if no exceptions occured).

INPUT:

- target_filename – the name of the file to be written. Normally, the contents of this file will be overwritten.
- append – (boolean, default: False) if True and target_filename is an existing file, then copy the current contents of target_filename to the temporary file when entering the with statement. Otherwise, the temporary file is initially empty.
- mode – (default: 0o666) mode bits for the file. The temporary file is created with mode mode & ~umask and the resulting file will also have these permissions (unless the mode bits of the file were changed manually).

EXAMPLES:

```
sage: from sage.misc.temporary_file import atomic_write
sage: target_file = tmp_filename()
sage: open(target_file, "w").write("Old contents")
sage: with atomic_write(target_file) as f:
....:     f.write("New contents")
....:     f.flush()
....:     open(target_file, "r").read()
'Old contents'
sage: open(target_file, "r").read()
'New contents'
```

The name of the temporary file can be accessed using f.name. It is not a problem to close and re-open the temporary file:

```
sage: from sage.misc.temporary_file import atomic_write
sage: target_file = tmp_filename()
sage: open(target_file, "w").write("Old contents")
sage: with atomic_write(target_file) as f:
```

```
....:     f.close()
....:     open(f.name, "w").write("Newer contents")
sage: open(target_file, "r").read()
'Newer contents'
```

If an exception occurs while writing the file, the target file is not touched:

```
sage: with atomic_write(target_file) as f:
....:     f.write("Newest contents")
....:     raise RuntimeError
Traceback (most recent call last):
...
RuntimeError
sage: open(target_file, "r").read()
'Newer contents'
```

Some examples of using the append option. Note that the file is never opened in "append" mode, it is possible to overwrite existing data:

```
sage: target_file = tmp_filename()
sage: with atomic_write(target_file, append=True) as f:
....:     f.write("Hello")
sage: with atomic_write(target_file, append=True) as f:
....:     f.write(" World")
sage: open(target_file, "r").read()
'Hello World'
sage: with atomic_write(target_file, append=True) as f:
....:     f.seek(0)
....:     f.write("HELLO")
sage: open(target_file, "r").read()
'HELLO World'
```

If the target file is a symbolic link, the link is kept and the target of the link is written to:

```
sage: link_to_target = os.path.join(tmp_dir(), "templink")
sage: os.symlink(target_file, link_to_target)
sage: with atomic_write(link_to_target) as f:
....:     f.write("Newest contents")
sage: open(target_file, "r").read()
'Newest contents'
```

We check the permission bits of the new file. Note that the old permissions do not matter:

```
sage: os.chmod(target_file, 0o600)
sage: _ = os.umask(0o022)
sage: with atomic_write(target_file) as f:
....:     pass
sage: oct(os.stat(target_file).st_mode & 0o777)
'644'
sage: _ = os.umask(0o077)
sage: with atomic_write(target_file, mode=0o777) as f:
....:     pass
sage: oct(os.stat(target_file).st_mode & 0o777)
'700'
```

Test writing twice to the same target file. The outermost with "wins":

```
sage: open(target_file, "w").write(">>> ")
sage: with atomic_write(target_file, append=True) as f,          ....:         atomic_write(targ
....:     f.write("AAA"); f.close()
```

```
....:     g.write("BBB"); g.close()
sage: open(target_file, "r").read()
'>>> AAA'
```

sage.misc.temporary_file.**delete_tmpfiles**()

Remove the directory SAGE_TMP.

TESTS:

This is automatically run when Sage exits, test this by running a separate session of Sage:

```
sage: from sage.tests.cmdline import test_executable
sage: child_SAGE_TMP, err, ret = test_executable(["sage", "-c", "print SAGE_TMP"])
sage: err, ret
('', 0)
sage: os.path.exists(child_SAGE_TMP)  # indirect doctest
False
```

The parent directory should exist:

```
sage: parent_SAGE_TMP = os.path.normpath(child_SAGE_TMP + '/..')
sage: os.path.isdir(parent_SAGE_TMP)
True
```

sage.misc.temporary_file.**graphics_filename**(*ext='.png'*)

Deprecated SageNB graphics filename

You should just use tmp_filename().

When run from the Sage notebook, return the next available canonical filename for a plot/graphics file in the current working directory. Otherwise, return a temporary file inside SAGE_TMP.

INPUT:

- ext – (default: ".png") A file extension (including the dot) for the filename.

OUTPUT:

The path of the temporary file created. In the notebook, this is a filename without path in the current directory. Otherwise, this an absolute path.

EXAMPLES:

```
sage: from sage.misc.temporary_file import graphics_filename
sage: print graphics_filename()  # random, typical filename for sagenb
sage0.png
```

TESTS:

When doctesting, this returns instead a random temporary file. We check that it's a file inside SAGE_TMP and that the extension is correct:

```
sage: fn = graphics_filename(ext=".jpeg")
sage: fn.startswith(str(SAGE_TMP))
True
sage: fn.endswith('.jpeg')
True
```

Historically, it was also possible to omit the dot. This has been changed in trac ticket #16640 but it will still work for now:

```
sage: fn = graphics_filename("jpeg")
doctest:...: DeprecationWarning: extension must now include the dot
```

```
See http://trac.sagemath.org/16640 for details.
sage: fn.endswith('.jpeg')
True
```

sage.misc.temporary_file.**tmp_dir**(*name='dir_'*, *ext=''*)

Create and return a temporary directory in `$HOME/.sage/temp/hostname/pid/`

The temporary directory is deleted automatically when Sage exits.

INPUT:

- `name` – (default: `"dir_"`) A prefix for the directory name.
- `ext` – (default: `""`) A suffix for the directory name.

OUTPUT:

The absolute path of the temporary directory created, with a trailing slash (or whatever the path separator is on your OS).

EXAMPLES:

```
sage: d = tmp_dir('dir_testing_', '.extension')
sage: d   # random output
'/home/username/.sage/temp/hostname/7961/dir_testing_XgRu4p.extension/'
sage: os.chdir(d)
sage: _ = open('file_inside_d', 'w')
```

Temporary directories are unaccessible by other users:

```
sage: os.stat(d).st_mode & 0o077
0
```

sage.misc.temporary_file.**tmp_filename**(*name='tmp_'*, *ext=''*)

Create and return a temporary file in `$HOME/.sage/temp/hostname/pid/`

The temporary file is deleted automatically when Sage exits.

> **Warning:** If you need a particular file extension always use `tmp_filename(ext=".foo")`, this will ensure that the file does not yet exist. If you were to use `tmp_filename()+".foo"`, then you might overwrite an existing file!

INPUT:

- `name` – (default: `"tmp_"`) A prefix for the file name.
- `ext` – (default: `""`) A suffix for the file name. If you want a filename extension in the usual sense, this should start with a dot.

OUTPUT:

The absolute path of the temporary file created.

EXAMPLES:

```
sage: fn = tmp_filename('just_for_testing_', '.extension')
sage: fn   # random
'/home/username/.sage/temp/hostname/8044/just_for_testing_tVVHsn.extension'
sage: _ = open(fn, 'w')
```

Temporary files are unaccessible by other users:

```
sage: os.stat(fn).st_mode & 0o077
0
```

CHAPTER

SIXTEEN

CONSTANT FUNCTIONS

class sage.misc.constant_function.**ConstantFunction**

Bases: sage.structure.sage_object.SageObject

A class for function objects implementing constant functions.

EXAMPLES:

```
sage: f = ConstantFunction(3)
sage: f
The constant function (...) -> 3
sage: f()
3
sage: f(5)
3
```

Such a function could be implemented as a lambda expression, but this is not (currently) picklable:

```
sage: g = lambda x: 3
sage: g == loads(dumps(g))
Traceback (most recent call last):
...
PicklingError: Can't pickle <type 'function'>: attribute lookup __builtin__.function failed
sage: f == loads(dumps(f))
True
```

Also, in the long run, the information that this function is constant could be used by some algorithms.

TODO:

- Should constant functions have unique representation?
- Should the number of arguments be specified in the input?
- Should this go into sage.categories.maps? Then what should be the parent (e.g. for lambda x: True)?

TESTS:

These tests do fail if we try to use UniqueRepresentation:

```
sage: f = ConstantFunction(True)
sage: g = ConstantFunction(1)
sage: f(), g()
(True, 1)
```

That's because 1 and True cannot be distinguished as keys in a dictionary (argl!):

```
sage: { 1: 'a', True: 'b' }
{1: 'b'}
```

CHAPTER

SEVENTEEN

SAGE PACKAGE MANAGEMENT COMMANDS

A Sage package has the extension .spkg. It is a tarball that is (usually) bzip2 compressed that contains arbitrary data and an spkg-install file. An Sage package typically has the following components:

- spkg-install - shell script that is run to install the package
- Sage.txt - file that describes how the package was made, who maintains it, etc.
- sage - directory with extra patched version of files that needed during the install

Use the `install_package` command to install a new package, and use `optional_packages` to list all optional packages available on the central Sage server. The `upgrade` command upgrades all *standard* packages - there is no auto-upgrade command for optional packages.

All package management can also be done via the Sage command line.

`sage.misc.package.`**`experimental_packages`**`()`

Return two lists. The first contains the installed and the second contains the not-installed experimental packages that are available from the Sage repository. You must have an internet connection.

OUTPUT:

- installed experimental packages (as a list)
- NOT installed experimental packages (as a list)

Use `install_package(package_name)` to install or re-install a given package.

See also:

`install_package()`, `upgrade()`

`sage.misc.package.`**`install_all_optional_packages`**`(`*force=True*, *dry_run=False*`)`

Install all available optional spkg's in the official Sage spkg repository. Returns a list of all spkg's that *fail* to install.

INPUT:

- `force` – bool (default: `True`); whether to force reinstall of spkg's that are already installed.
- `dry_run` – bool (default: `False`); if `True`, just list the packages that would be installed in order, but don't actually install them.

OUTPUT:

list of strings

Note: This is designed mainly for testing purposes. This also doesn't do anything with respect to dependencies – the packages are installed in alphabetical order. Dependency issues will be dealt with in a future version.

AUTHOR:

– William Stein (2008-12)

EXAMPLES:

```
sage: sage.misc.package.install_all_optional_packages(dry_run=True)  # optional - internet
Installing ...
[]
```

`sage.misc.package.install_package(package=None, force=False)`

Install a package or return a list of all packages that have been installed into this Sage install.

You must have an internet connection. Also, you will have to restart Sage for the changes to take affect.

It is not needed to provide the version number.

INPUT:

- `package` – optional; if specified, install the given package. If not, list all installed packages.
- `force` – boolean (default: `False`); if `True`, reinstall the package given if it is already installed. (Otherwise, an already installed package doesn't get reinstalled, as with 'sage -i ...'.)

EXAMPLES:

With no arguments, list the installed packages:

```
sage: install_package()
[...'atlas...'python...]
```

With an argument, install the named package:

```
sage: install_package('chomp')  # not tested
Attempting to download package chomp-20100213.p2
...
```

IMPLEMENTATION:

Calls 'sage -f ...' to (re)install the package if a package name is given. If no package name is given, simply list the contents of `spkg/installed`.

See also:

`optional_packages()`, `upgrade()`

`sage.misc.package.is_package_installed(package)`

Return true if a package starting with the given string is installed.

EXAMPLES:

```
sage: is_package_installed('sage')
True
```

`sage.misc.package.optional_packages()`

Return two lists. The first contains the installed and the second contains the not-installed optional packages that are available from the Sage repository. You must have an internet connection.

OUTPUT:

- installed optional packages (as a list)
- NOT installed optional packages (as a list)

Use `install_package(package_name)` to install or re-install a given package.

See also:

`install_package()`, `upgrade()`

`sage.misc.package.`**`package_mesg`**`(`*package_name*`)`

x.__init__(...) initializes x; see help(type(x)) for signature

`sage.misc.package.`**`standard_packages`**`()`

Return two lists. The first contains the installed and the second contains the not-installed standard packages that are available from the Sage repository. You must have an internet connection.

OUTPUT:

- installed standard packages (as a list)
- NOT installed standard packages (as a list)

Use `install_package(package_name)` to install or re-install a given package.

See also:

`install_package()`, `upgrade()`

`sage.misc.package.`**`upgrade`**`()`

Download and build the latest version of Sage.

You must have an internet connection. Also, you will have to restart Sage for the changes to take affect.

This upgrades to the latest version of core packages (optional packages are not automatically upgraded).

This will not work on systems that don't have a C compiler.

See also:

`install_package()`, `optional_packages()`

CHAPTER EIGHTEEN

A TOOL FOR INSPECTING PYTHON PICKLES

AUTHORS:

- Carl Witty (2009-03)

The explain_pickle function takes a pickle and produces Sage code that will evaluate to the contents of the pickle. Ideally, the combination of explain_pickle to produce Sage code and sage_eval to evaluate the code would be a 100% compatible implementation of cPickle's unpickler; this is almost the case now.

EXAMPLES:

```
sage: explain_pickle(dumps(12345))
pg_make_integer = unpickle_global('sage.rings.integer', 'make_integer')
pg_make_integer('c1p')
sage: explain_pickle(dumps(polygen(QQ)))
pg_Polynomial_rational_flint = unpickle_global('sage.rings.polynomial.polynomial_rational_flint', 'Po
pg_PolynomialRing = unpickle_global('sage.rings.polynomial.polynomial_ring_constructor', 'PolynomialR
pg_RationalField = unpickle_global('sage.rings.rational_field', 'RationalField')
pg = unpickle_instantiate(pg_RationalField, ())
pg_make_rational = unpickle_global('sage.rings.rational', 'make_rational')
pg_Polynomial_rational_flint(pg_PolynomialRing(pg, 'x', None, False), [pg_make_rational('0'), pg_make
sage: sage_eval(explain_pickle(dumps(polygen(QQ)))) == polygen(QQ)
True
```

By default (as above) the code produced contains calls to several utility functions (unpickle_global, etc.); this is done so that the code is truly equivalent to the pickle. If the pickle can be loaded into a future version of Sage, then the code that explain_pickle produces today should work in that future Sage as well.

It is also possible to produce simpler code, that is tied to the current version of Sage; here are the above two examples again:

```
sage: explain_pickle(dumps(12345), in_current_sage=True)
from sage.rings.integer import make_integer
make_integer('c1p')
sage: explain_pickle(dumps(polygen(QQ)), in_current_sage=True)
from sage.rings.polynomial.polynomial_rational_flint import Polynomial_rational_flint
from sage.rings.rational import make_rational
Polynomial_rational_flint(PolynomialRing(RationalField(), 'x', None, False), [make_rational('0'), mak
```

The explain_pickle function has several use cases.

- Write pickling support for your classes

  You can use explain_pickle to see what will happen when a pickle is unpickled. Consider: is this sequence of commands something that can be easily supported in all future Sage versions, or does it expose internal design decisions that are subject to change?

- Debug old pickles

  If you have a pickle from an old version of Sage that no longer unpickles, you can use explain_pickle to see what it is trying to do, to figure out how to fix it.

- Use explain_pickle in doctests to help maintenance

  If you have a `loads(dumps(S))` doctest, you could also add an `explain_pickle(dumps(S))` doctest. Then if something changes in a way that would invalidate old pickles, the output of `explain_pickle` will also change. At that point, you can add the previous output of `explain_pickle` as a new set of doctests (and then update the :obj‘explain_pickle‘ doctest to use the new output), to ensure that old pickles will continue to work. (These problems will also be caught using the `picklejar`, but having the tests directly in the relevant module is clearer.)

As mentioned above, there are several output modes for `explain_pickle`, that control fidelity versus simplicity of the output. For example, the GLOBAL instruction takes a module name and a class name and produces the corresponding class. So GLOBAL of `sage.rings.integer`, `Integer` is approximately equivalent to `sage.rings.integer.Integer`.

However, this class lookup process can be customized (using sage.structure.sage_object.register_unpickle_override). For instance, if some future version of Sage renamed `sage/rings/integer.pyx` to `sage/rings/knuth_was_here.pyx`, old pickles would no longer work unless register_unpickle_override was used; in that case, GLOBAL of ‘sage.rings.integer’, ‘integer’ would mean `sage.rings.knuth_was_here.integer`.

By default, `explain_pickle` will map this GLOBAL instruction to `unpickle_global('sage.rings.integer', 'integer')`. Then when this code is evaluated, unpickle_global will look up the current mapping in the register_unpickle_override table, so the generated code will continue to work even in hypothetical future versions of Sage where integer.pyx has been renamed.

If you pass the flag `in_current_sage=True`, then `explain_pickle` will generate code that may only work in the current version of Sage, not in future versions. In this case, it would generate:

```
from sage.rings.integer import integer
```

and if you ran explain_pickle in hypothetical future sage, it would generate:

> from sage.rings.knuth_was_here import integer

but the current code wouldn’t work in the future sage.

If you pass the flag `default_assumptions=True`, then `explain_pickle` will generate code that would work in the absence of any special unpickling information. That is, in either current Sage or hypothetical future Sage, it would generate:

```
from sage.rings.integer import integer
```

The intention is that `default_assumptions` output is prettier (more human-readable), but may not actually work; so it is only intended for human reading.

There are several functions used in the output of `explain_pickle`. Here I give a brief description of what they usually do, as well as how to modify their operation (for instance, if you’re trying to get old pickles to work).

- `unpickle_global(module, classname)`: unpickle_global(‘sage.foo.bar’, ‘baz’) is usually equivalent to sage.foo.bar.baz, but this can be customized with register_unpickle_override.
- `unpickle_newobj(klass, args)`: Usually equivalent to `klass.__new__(klass, *args)`. If `klass` is a Python class, then you can define `__new__()` to control the result (this result actually need not be an instance of klass). (This doesn’t work for Cython classes.)

- `unpickle_build(obj, state)`: If `obj` has a `__setstate__()` method, then this is equivalent to `obj.__setstate__(state)`. Otherwise uses state to set the attributes of `obj`. Customize by defining `__setstate__()`.
- `unpickle_instantiate(klass, args)`: Usually equivalent to `klass(*args)`. Cannot be customized.
- unpickle_appends(lst, vals): Appends the values in vals to lst. If not `isinstance(lst, list)`, can be customized by defining a `append()` method.

class `sage.misc.explain_pickle.`**`EmptyNewstyleClass`**

Bases: `object`

A featureless new-style class (inherits from object); used for testing explain_pickle.

class `sage.misc.explain_pickle.`**`EmptyOldstyleClass`**

A featureless old-style class (does not inherit from object); used for testing explain_pickle.

class `sage.misc.explain_pickle.`**`PickleDict`**(*items*)

Bases: `object`

An object which can be used as the value of a PickleObject. The items is a list of key-value pairs, where the keys and values are SageInputExpressions. We use this to help construct dictionary literals, instead of always starting with an empty dictionary and assigning to it.

class `sage.misc.explain_pickle.`**`PickleExplainer`**(*sib*, *in_current_sage=False*, *default_assumptions=False*, *pedantic=False*)

Bases: `object`

An interpreter for the pickle virtual machine, that executes symbolically and constructs SageInputExpressions instead of directly constructing values.

APPEND()

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(['a'])
    0: \x80 PROTO      2
    2: ]    EMPTY_LIST
    3: q    BINPUT     1
    5: U    SHORT_BINSTRING 'a'
    8: a    APPEND
    9: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
['a']
result: ['a']
```

As shown above, we prefer to create a list literal. This is not possible if the list is recursive:

```
sage: v = []
sage: v.append(v)
sage: test_pickle(v)
    0: \x80 PROTO      2
    2: ]    EMPTY_LIST
    3: q    BINPUT     1
    5: h    BINGET     1
    7: a    APPEND
    8: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
si = []
```

```
list.append(si, si)
si
result: [[...]]
```

APPENDS ()

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(['a', 'b'])
    0: \x80 PROTO      2
    2: ]    EMPTY_LIST
    3: q    BINPUT     1
    5: (    MARK
    6: U        SHORT_BINSTRING 'a'
    9: U        SHORT_BINSTRING 'b'
   12: e        APPENDS    (MARK at 5)
   13: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
['a', 'b']
result: ['a', 'b']
```

As shown above, we prefer to create a list literal. This is not possible if the list is recursive:

```
sage: v = []
sage: v.append(v)
sage: v.append(v)
sage: test_pickle(v)
    0: \x80 PROTO      2
    2: ]    EMPTY_LIST
    3: q    BINPUT     1
    5: (    MARK
    6: h        BINGET     1
    8: h        BINGET     1
   10: e        APPENDS    (MARK at 5)
   11: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
si = []
list.extend(si, [si, si])
si
result: [[...], [...]]
```

BINFLOAT (*f*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(float(pi))
    0: \x80 PROTO      2
    2: G    BINFLOAT   3.141592653589793
   11: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
float(RR(3.1415926535897931))
result: 3.141592653589793
```

BINGET (*n*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_LIST + BINPUT + 'x' + POP + BINGET + 'x' + '.')
    0: ]    EMPTY_LIST
    1: q    BINPUT     120
    3: 0    POP
    4: h    BINGET     120
    6: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
[]
result: []
```

BININT (*n*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(dumps(100000r, compress=False))
    0: \x80 PROTO      2
    2: J    BININT     100000
    7: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
100000
result: 100000
```

BININT1 (*n*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(dumps(100r, compress=False))
    0: \x80 PROTO      2
    2: K    BININT1    100
    4: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
100
result: 100
```

BININT2 (*n*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(dumps(1000r, compress=False))
    0: \x80 PROTO      2
    2: M    BININT2    1000
    5: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
1000
result: 1000
```

BINPERSID ()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(INT + "0\n" + BINPERSID + '.', args=('Yo!',))
    0: I    INT        0
```

```
        3: Q    BINPERSID
        4: .    STOP
    highest protocol among opcodes = 1
    explain_pickle in_current_sage=True/False:
    unpickle_persistent(0)
    result: 'Yo!'
```

BINPUT (*n*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_LIST + BINPUT + 'x' + POP + BINGET + 'x')
    0: ]    EMPTY_LIST
    1: q    BINPUT     120
    3: 0    POP
    4: h    BINGET     120
    6: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
[]
result: []
```

BINSTRING (*s*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle('T\5\0\0\0hello.')
    0: T    BINSTRING 'hello'
   10: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
'hello'
result: 'hello'
```

BINUNICODE (*s*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(u'hi\u1234\U00012345')
    0: \x80 PROTO      2
    2: X    BINUNICODE u'hi\u1234\U00012345'
   16: q    BINPUT     1
   18: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
u'hi\u1234\U00012345'
result: u'hi\u1234\U00012345'
```

BUILD ()

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(TestBuild())
    0: \x80 PROTO      2
    2: c    GLOBAL     'sage.misc.explain_pickle TestBuild'
   38: q    BINPUT     1
   40: )    EMPTY_TUPLE
   41: \x81 NEWOBJ
```

```
   42: q    BINPUT     2
   44: }    EMPTY_DICT
   45: q    BINPUT     3
   47: U    SHORT_BINSTRING 'x'
   50: K    BININT1    3
   52: s    SETITEM
   53: }    EMPTY_DICT
   54: q    BINPUT     4
   56: U    SHORT_BINSTRING 'y'
   59: K    BININT1    4
   61: s    SETITEM
   62: \x86 TUPLE2
   63: b    BUILD
   64: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True:
from sage.misc.explain_pickle import TestBuild
si = unpickle_newobj(TestBuild, ())
si.__dict__['x'] = 3
si.y = 4
si
explain_pickle in_current_sage=False:
pg_TestBuild = unpickle_global('sage.misc.explain_pickle', 'TestBuild')
si = unpickle_newobj(pg_TestBuild, ())
unpickle_build(si, ({'x':3}, {'y':4}))
si
result: TestBuild: x=3; y=4

sage: test_pickle(TestBuildSetstate(), verbose_eval=True)
    0: \x80 PROTO      2
    2: c    GLOBAL     'sage.misc.explain_pickle TestBuildSetstate'
   46: q    BINPUT     1
   48: )    EMPTY_TUPLE
   49: \x81 NEWOBJ
   50: q    BINPUT     2
   52: }    EMPTY_DICT
   53: q    BINPUT     3
   55: U    SHORT_BINSTRING 'x'
   58: K    BININT1    3
   60: s    SETITEM
   61: }    EMPTY_DICT
   62: q    BINPUT     4
   64: U    SHORT_BINSTRING 'y'
   67: K    BININT1    4
   69: s    SETITEM
   70: \x86 TUPLE2
   71: b    BUILD
   72: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True:
from sage.misc.explain_pickle import TestBuildSetstate
si = unpickle_newobj(TestBuildSetstate, ())
si.__setstate__(({'x':3}, {'y':4}))
si
explain_pickle in_current_sage=False:
pg_TestBuildSetstate = unpickle_global('sage.misc.explain_pickle', 'TestBuildSetstate')
si = unpickle_newobj(pg_TestBuildSetstate, ())
unpickle_build(si, ({'x':3}, {'y':4}))
```

```
si
evaluating explain_pickle in_current_sage=True:
setting state from ({'x': 3}, {'y': 4})
evaluating explain_pickle in_current_sage=False:
setting state from ({'x': 3}, {'y': 4})
loading pickle with cPickle:
setting state from ({'x': 3}, {'y': 4})
result: TestBuild: x=4; y=3
```

DICT()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(DICT, args=('mark', 'a', 1, 2, 'b'))
    0: (    MARK
    1: P        PERSID     '1'
    4: P        PERSID     '2'
    7: P        PERSID     '3'
   10: P        PERSID     '4'
   13: d        DICT       (MARK at 0)
   14: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
{unpickle_persistent('1'):unpickle_persistent('2'), unpickle_persistent('3'):unpickle_persis
result: {'a': 1, 2: 'b'}
```

DUP()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_LIST + DUP + TUPLE2 + STOP)
    0: ]    EMPTY_LIST
    1: 2    DUP
    2: \x86 TUPLE2
    3: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
si = []
(si, si)
result: ([], [])
```

EMPTY_DICT()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_DICT)
    0: }    EMPTY_DICT
    1: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
{}
result: {}
```

EMPTY_LIST()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_LIST)
    0: ]    EMPTY_LIST
    1: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
[]
result: []
```

EMPTY_TUPLE()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_TUPLE)
    0: )    EMPTY_TUPLE
    1: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
()
result: ()
```

EXT1(*n*)

TESTS:

```
sage: from copy_reg import *
sage: from sage.misc.explain_pickle import *
sage: add_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 42)
sage: test_pickle(EmptyNewstyleClass())
    0: \x80 PROTO      2
    2: \x82 EXT1       42
    4: )    EMPTY_TUPLE
    5: \x81 NEWOBJ
    6: q    BINPUT     1
    8: }    EMPTY_DICT
    9: q    BINPUT     2
   11: b    BUILD
   12: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
si = unpickle_newobj(unpickle_extension(42), ())
unpickle_build(si, {})
si
result: EmptyNewstyleClass
sage: remove_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 42)
```

EXT2(*n*)

TESTS:

```
sage: from copy_reg import *
sage: from sage.misc.explain_pickle import *
sage: add_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 31415)
sage: test_pickle(EmptyNewstyleClass())
    0: \x80 PROTO      2
    2: \x83 EXT2       31415
    5: )    EMPTY_TUPLE
    6: \x81 NEWOBJ
    7: q    BINPUT     1
```

```
    9: }     EMPTY_DICT
   10: q     BINPUT     2
   12: b     BUILD
   13: .     STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
si = unpickle_newobj(unpickle_extension(31415), ())
unpickle_build(si, {})
si
result: EmptyNewstyleClass
sage: remove_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 31415)
```

EXT4 (*n*)

TESTS:

```
sage: from copy_reg import *
sage: from sage.misc.explain_pickle import *
sage: add_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 27182818)
sage: test_pickle(EmptyNewstyleClass())
    0: \x80 PROTO      2
    2: \x84 EXT4       27182818
    7: )    EMPTY_TUPLE
    8: \x81 NEWOBJ
    9: q    BINPUT     1
   11: }    EMPTY_DICT
   12: q    BINPUT     2
   14: b    BUILD
   15: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
si = unpickle_newobj(unpickle_extension(27182818), ())
unpickle_build(si, {})
si
result: EmptyNewstyleClass
sage: remove_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 27182818)
```

FLOAT (*f*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(FLOAT + '2.71828\n')
    0: F    FLOAT      2.71828
    9: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
2.71828
result: 2.71828
```

GET (*n*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_LIST + PUT + '1\n' + POP + GET + '1\n' + '.')
    0: ]    EMPTY_LIST
    1: p    PUT        1
    4: 0    POP
    5: g    GET        1
```

```
    8: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
[]
result: []
```

GLOBAL (*name*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
```

We've used register_unpickle_override so that unpickle_global will map TestGlobalOldName to TestGlobalNewName.

```
sage: test_pickle(TestGlobalOldName())
    0: \x80 PROTO      2
    2: c    GLOBAL     'sage.misc.explain_pickle TestGlobalOldName'
   46: q    BINPUT     1
   48: )    EMPTY_TUPLE
   49: \x81 NEWOBJ
   50: q    BINPUT     2
   52: }    EMPTY_DICT
   53: q    BINPUT     3
   55: b    BUILD
   56: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True:
from sage.misc.explain_pickle import TestGlobalNewName
unpickle_newobj(TestGlobalNewName, ())
explain_pickle in_current_sage=False:
pg_TestGlobalOldName = unpickle_global('sage.misc.explain_pickle', 'TestGlobalOldName')
si = unpickle_newobj(pg_TestGlobalOldName, ())
unpickle_build(si, {})
si
result: TestGlobalNewName
```

Note that default_assumptions blithely assumes that it should use the old name, giving code that doesn't actually work as desired:

```
sage: explain_pickle(dumps(TestGlobalOldName()), default_assumptions=True)
from sage.misc.explain_pickle import TestGlobalOldName
unpickle_newobj(TestGlobalOldName, ())
```

A class name need not be a valid identifier:

```
sage: sage.misc.explain_pickle.__dict__['funny$name'] = TestGlobalFunnyName # see comment at
sage: test_pickle((TestGlobalFunnyName(), TestGlobalFunnyName()))
    0: \x80 PROTO      2
    2: c    GLOBAL     'sage.misc.explain_pickle funny$name'
   39: q    BINPUT     1
   41: )    EMPTY_TUPLE
   42: \x81 NEWOBJ
   43: q    BINPUT     2
   45: }    EMPTY_DICT
   46: q    BINPUT     3
   48: b    BUILD
   49: h    BINGET     1
   51: )    EMPTY_TUPLE
   52: \x81 NEWOBJ
```

```
       53: q    BINPUT     4
       55: }    EMPTY_DICT
       56: q    BINPUT     5
       58: b    BUILD
       59: \x86 TUPLE2
       60: q    BINPUT     6
       62: .    STOP
    highest protocol among opcodes = 2
    explain_pickle in_current_sage=True/False:
    si1 = unpickle_global('sage.misc.explain_pickle', 'funny$name')
    si2 = unpickle_newobj(si1, ())
    unpickle_build(si2, {})
    si3 = unpickle_newobj(si1, ())
    unpickle_build(si3, {})
    (si2, si3)
    result: (TestGlobalFunnyName, TestGlobalFunnyName)
```

INST (*name*)

TESTS:

```
sage: import pickle
sage: from sage.misc.explain_pickle import *
sage: test_pickle(pickle.dumps(EmptyOldstyleClass(), protocol=0))
    0: (    MARK
    1: i        INST       'sage.misc.explain_pickle EmptyOldstyleClass' (MARK at 0)
   46: p    PUT        0
   49: (    MARK
   50: d        DICT       (MARK at 49)
   51: p    PUT        1
   54: b    BUILD
   55: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True:
from types import InstanceType
from sage.misc.explain_pickle import EmptyOldstyleClass
InstanceType(EmptyOldstyleClass)
explain_pickle in_current_sage=False:
pg_EmptyOldstyleClass = unpickle_global('sage.misc.explain_pickle', 'EmptyOldstyleClass')
pg = unpickle_instantiate(pg_EmptyOldstyleClass, ())
unpickle_build(pg, {})
pg
result: EmptyOldstyleClass
```

INT (*n*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(INT + "-12345\n")
    0: I    INT        -12345
    8: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
-12345
result: -12345
```

INT can also be used to record True and False:

```
sage: test_pickle(INT + "00\n")
    0: I    INT        False
    4: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
False
result: False
sage: test_pickle(INT + "01\n")
    0: I    INT        True
    4: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
True
result: True
```

LIST()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(MARK + NONE + NEWFALSE + LIST)
    0: (    MARK
    1: N        NONE
    2: \x89     NEWFALSE
    3: l        LIST       (MARK at 0)
    4: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
[None, False]
result: [None, False]
```

LONG(*n*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(LONG + "12345678909876543210123456789L\n")
    0: L    LONG       12345678909876543210123456789L
   32: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
12345678909876543210123456789
result: 12345678909876543210123456789L
```

LONG1(*n*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(1L)
    0: \x80 PROTO      2
    2: \x8a LONG1      1L
    5: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
1L
result: 1L
```

LONG4(*n*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(LONG4 + '\014\0\0\0' + 'hello, world')
    0: \x8b LONG4      31079605376604435891501163880L
   17: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
31079605376604435891501163880
result: 31079605376604435891501163880L
```

LONG_BINGET(*n*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_LIST + LONG_BINPUT + 'Sage' + POP + LONG_BINGET + 'Sage')
    0: ]    EMPTY_LIST
    1: r    LONG_BINPUT 1701273939
    6: 0    POP
    7: j    LONG_BINGET 1701273939
   12: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
[]
result: []
```

LONG_BINPUT(*n*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_LIST + LONG_BINPUT + 'Sage' + POP + LONG_BINGET + 'Sage')
    0: ]    EMPTY_LIST
    1: r    LONG_BINPUT 1701273939
    6: 0    POP
    7: j    LONG_BINGET 1701273939
   12: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
[]
result: []
```

MARK()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(MARK + TUPLE)
    0: (    MARK
    1: t        TUPLE      (MARK at 0)
    2: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
()
result: ()
```

NEWFALSE()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(NEWFALSE)
    0: \x89 NEWFALSE
    1: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
False
result: False
```

NEWOBJ()

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EmptyNewstyleClass())
    0: \x80 PROTO      2
    2: c    GLOBAL     'sage.misc.explain_pickle EmptyNewstyleClass'
   47: q    BINPUT     1
   49: )    EMPTY_TUPLE
   50: \x81 NEWOBJ
   51: q    BINPUT     2
   53: }    EMPTY_DICT
   54: q    BINPUT     3
   56: b    BUILD
   57: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True:
from sage.misc.explain_pickle import EmptyNewstyleClass
unpickle_newobj(EmptyNewstyleClass, ())
explain_pickle in_current_sage=False:
pg_EmptyNewstyleClass = unpickle_global('sage.misc.explain_pickle', 'EmptyNewstyleClass')
si = unpickle_newobj(pg_EmptyNewstyleClass, ())
unpickle_build(si, {})
si
result: EmptyNewstyleClass
```

NEWTRUE()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(NEWTRUE)
    0: \x88 NEWTRUE
    1: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
True
result: True
```

NONE()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(NONE)
    0: N    NONE
    1: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
```

```
None
result: None
```

OBJ()

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EmptyOldstyleClass())
    0: \x80 PROTO      2
    2: (    MARK
    3: c        GLOBAL     'sage.misc.explain_pickle EmptyOldstyleClass'
   48: q        BINPUT     1
   50: o        OBJ        (MARK at 2)
   51: q    BINPUT     2
   53: }    EMPTY_DICT
   54: q    BINPUT     3
   56: b    BUILD
   57: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True:
from types import InstanceType
from sage.misc.explain_pickle import EmptyOldstyleClass
InstanceType(EmptyOldstyleClass)
explain_pickle in_current_sage=False:
pg_EmptyOldstyleClass = unpickle_global('sage.misc.explain_pickle', 'EmptyOldstyleClass')
pg = unpickle_instantiate(pg_EmptyOldstyleClass, ())
unpickle_build(pg, {})
pg
result: EmptyOldstyleClass
```

PERSID(*id*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(PERSID + "0\n" + '.', args=('Yo!',))
    0: P    PERSID     '0'
    3: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
unpickle_persistent('0')
result: 'Yo!'
```

POP()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(INT + "0\n" + POP + INT + "42\n")
    0: I    INT        0
    3: 0    POP
    4: I    INT        42
    8: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
42
result: 42
```

POP_MARK()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(MARK + NONE + NEWFALSE + POP_MARK + NEWTRUE)
    0: (    MARK
    1: N        NONE
    2: \x89     NEWFALSE
    3: 1        POP_MARK   (MARK at 0)
    4: \x88 NEWTRUE
    5: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
True
result: True
```

PROTO (*proto*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(0r)
    0: \x80 PROTO      2
    2: K    BININT1    0
    4: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
0
result: 0
```

PUT (*n*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_LIST + PUT + '1\n' + POP + GET + '1\n' + '.')
    0: ]    EMPTY_LIST
    1: p    PUT        1
    4: 0    POP
    5: g    GET        1
    8: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
[]
result: []
```

REDUCE ()

TESTS:

```
sage: import pickle
sage: from sage.misc.explain_pickle import *
sage: test_pickle(pickle.dumps(EmptyNewstyleClass(), protocol=1))
    0: c    GLOBAL     'copy_reg _reconstructor'
   25: q    BINPUT     0
   27: (    MARK
   28: c        GLOBAL     'sage.misc.explain_pickle EmptyNewstyleClass'
   73: q        BINPUT     1
   75: c        GLOBAL     '__builtin__ object'
   95: q        BINPUT     2
   97: N        NONE
   98: t        TUPLE      (MARK at 27)
```

```
   99: q    BINPUT     3
  101: R    REDUCE
  102: q    BINPUT     4
  104: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True:
from copy_reg import _reconstructor
from sage.misc.explain_pickle import EmptyNewstyleClass
from __builtin__ import object
_reconstructor(EmptyNewstyleClass, object, None)
explain_pickle in_current_sage=False:
pg__reconstructor = unpickle_global('copy_reg', '_reconstructor')
pg_EmptyNewstyleClass = unpickle_global('sage.misc.explain_pickle', 'EmptyNewstyleClass')
pg_object = unpickle_global('__builtin__', 'object')
pg__reconstructor(pg_EmptyNewstyleClass, pg_object, None)
result: EmptyNewstyleClass

sage: test_pickle(TestReduceGetinitargs(), verbose_eval=True)
Running __init__ for TestReduceGetinitargs
    0: \x80 PROTO      2
    2: (    MARK
    3: c        GLOBAL     'sage.misc.explain_pickle TestReduceGetinitargs'
   51: q        BINPUT     1
   53: o        OBJ        (MARK at 2)
   54: q    BINPUT     2
   56: }    EMPTY_DICT
   57: q    BINPUT     3
   59: b    BUILD
   60: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True:
from sage.misc.explain_pickle import TestReduceGetinitargs
TestReduceGetinitargs()
explain_pickle in_current_sage=False:
pg_TestReduceGetinitargs = unpickle_global('sage.misc.explain_pickle', 'TestReduceGetinitarg
pg = unpickle_instantiate(pg_TestReduceGetinitargs, ())
unpickle_build(pg, {})
pg
evaluating explain_pickle in_current_sage=True:
Running __init__ for TestReduceGetinitargs
evaluating explain_pickle in_current_sage=False:
Running __init__ for TestReduceGetinitargs
loading pickle with cPickle:
Running __init__ for TestReduceGetinitargs
result: TestReduceGetinitargs

sage: test_pickle(TestReduceNoGetinitargs(), verbose_eval=True)
Running __init__ for TestReduceNoGetinitargs
    0: \x80 PROTO      2
    2: (    MARK
    3: c        GLOBAL     'sage.misc.explain_pickle TestReduceNoGetinitargs'
   53: q        BINPUT     1
   55: o        OBJ        (MARK at 2)
   56: q    BINPUT     2
   58: }    EMPTY_DICT
   59: q    BINPUT     3
   61: b    BUILD
   62: .    STOP
```

```
highest protocol among opcodes = 2
explain_pickle in_current_sage=True:
from types import InstanceType
from sage.misc.explain_pickle import TestReduceNoGetinitargs
InstanceType(TestReduceNoGetinitargs)
explain_pickle in_current_sage=False:
pg_TestReduceNoGetinitargs = unpickle_global('sage.misc.explain_pickle', 'TestReduceNoGetini
pg = unpickle_instantiate(pg_TestReduceNoGetinitargs, ())
unpickle_build(pg, {})
pg
evaluating explain_pickle in_current_sage=True:
evaluating explain_pickle in_current_sage=False:
loading pickle with cPickle:
result: TestReduceNoGetinitargs
```

SETITEM()

TESTS:

```
sage: import pickle
sage: from sage.misc.explain_pickle import *
sage: test_pickle(pickle.dumps({'a': 'b'}))
    0: (    MARK
    1: d        DICT       (MARK at 0)
    2: p    PUT        0
    5: S    STRING     'a'
   10: p    PUT        1
   13: S    STRING     'b'
   18: p    PUT        2
   21: s    SETITEM
   22: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
{'a':'b'}
result: {'a': 'b'}
```

We see above that we output the result as a dictionary literal, when possible. This is impossible when a key or value is recursive. First we test recursive values:

```
sage: value_rec = dict()
sage: value_rec['circular'] = value_rec
sage: test_pickle(pickle.dumps(value_rec))
    0: (    MARK
    1: d        DICT       (MARK at 0)
    2: p    PUT        0
    5: S    STRING     'circular'
   17: p    PUT        1
   20: g    GET        0
   23: s    SETITEM
   24: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
si = {}
si['circular'] = si
si
result: {'circular': {...}}
```

Then we test recursive keys:

```
sage: key_rec = dict()
sage: key = EmptyNewstyleClass()
sage: key.circular = key_rec
sage: key_rec[key] = 'circular'
sage: test_pickle(pickle.dumps(key_rec))
    0: (    MARK
    1: d        DICT       (MARK at 0)
    2: p    PUT        0
    5: c    GLOBAL     'copy_reg _reconstructor'
   30: p    PUT        1
   33: (    MARK
   34: c        GLOBAL     'sage.misc.explain_pickle EmptyNewstyleClass'
   79: p        PUT        2
   82: c        GLOBAL     '__builtin__ object'
  102: p        PUT        3
  105: N        NONE
  106: t        TUPLE      (MARK at 33)
  107: p    PUT        4
  110: R    REDUCE
  111: p    PUT        5
  114: (    MARK
  115: d        DICT       (MARK at 114)
  116: p    PUT        6
  119: S    STRING     'circular'
  131: p    PUT        7
  134: g    GET        0
  137: s    SETITEM
  138: b    BUILD
  139: g    GET        7
  142: s    SETITEM
  143: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True:
si1 = {}
from copy_reg import _reconstructor
from sage.misc.explain_pickle import EmptyNewstyleClass
from __builtin__ import object
si2 = _reconstructor(EmptyNewstyleClass, object, None)
si2.__dict__['circular'] = si1
si1[si2] = 'circular'
si1
explain_pickle in_current_sage=False:
si1 = {}
pg__reconstructor = unpickle_global('copy_reg', '_reconstructor')
pg_EmptyNewstyleClass = unpickle_global('sage.misc.explain_pickle', 'EmptyNewstyleClass')
pg_object = unpickle_global('__builtin__', 'object')
si2 = pg__reconstructor(pg_EmptyNewstyleClass, pg_object, None)
unpickle_build(si2, {'circular':si1})
si1[si2] = 'circular'
si1
result: {EmptyNewstyleClass: 'circular'}
```

SETITEMS()

TESTS:

```
sage: import pickle
sage: from sage.misc.explain_pickle import *
sage: test_pickle(pickle.dumps({'a': 'b', 1r : 2r}, protocol=2))
```

```
    0: \x80 PROTO      2
    2: }    EMPTY_DICT
    3: q    BINPUT     0
    5: (    MARK
    6: U        SHORT_BINSTRING 'a'
    9: q        BINPUT     1
   11: U        SHORT_BINSTRING 'b'
   14: q        BINPUT     2
   16: K        BININT1    1
   18: K        BININT1    2
   20: u        SETITEMS   (MARK at 5)
   21: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
{'a':'b', 1:2}
result: {'a': 'b', 1: 2}
```

Similar to the tests for SETITEM, we test recursive keys and values:

```
sage: recdict = {}
sage: recdict['Circular value'] = recdict
sage: key = EmptyOldstyleClass()
sage: key.recdict = recdict
sage: recdict[key] = 'circular_key'
sage: test_pickle(pickle.dumps(recdict, protocol=2))
    0: \x80 PROTO      2
    2: }    EMPTY_DICT
    3: q    BINPUT     0
    5: (    MARK
    6: (        MARK
    7: c            GLOBAL     'sage.misc.explain_pickle EmptyOldstyleClass'
   52: q            BINPUT     1
   54: o            OBJ        (MARK at 6)
   55: q        BINPUT     2
   57: }        EMPTY_DICT
   58: q        BINPUT     3
   60: U        SHORT_BINSTRING 'recdict'
   69: q        BINPUT     4
   71: h        BINGET     0
   73: s        SETITEM
   74: b        BUILD
   75: U        SHORT_BINSTRING 'circular_key'
   89: q        BINPUT     5
   91: U        SHORT_BINSTRING 'Circular value'
  107: q        BINPUT     6
  109: h        BINGET     0
  111: u        SETITEMS   (MARK at 5)
  112: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True:
si1 = {}
from types import InstanceType
from sage.misc.explain_pickle import EmptyOldstyleClass
si2 = InstanceType(EmptyOldstyleClass)
si2.__dict__['recdict'] = si1
si1[si2] = 'circular_key'
si1['Circular value'] = si1
si1
explain_pickle in_current_sage=False:
```

```
si = {}
pg_EmptyOldstyleClass = unpickle_global('sage.misc.explain_pickle', 'EmptyOldstyleClass')
pg = unpickle_instantiate(pg_EmptyOldstyleClass, ())
unpickle_build(pg, {'recdict':si})
si[pg] = 'circular_key'
si['Circular value'] = si
si
result: {EmptyOldstyleClass: 'circular_key', 'Circular value': {...}}
```

SHORT_BINSTRING (*s*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(dumps('hello', compress=False))
    0: \x80 PROTO      2
    2: U    SHORT_BINSTRING 'hello'
    9: q    BINPUT     1
   11: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
'hello'
result: 'hello'
```

STOP ()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_TUPLE)
    0: )    EMPTY_TUPLE
    1: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
()
result: ()
```

STRING (*s*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle("S'Testing...'\n.")
    0: S    STRING     'Testing...'
   14: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
'Testing...'
result: 'Testing...'
```

TUPLE ()

TESTS:

```
sage: import pickle
sage: from sage.misc.explain_pickle import *
sage: test_pickle(pickle.dumps(('a',)))
    0: (    MARK
    1: S        STRING     'a'
    6: p        PUT        0
    9: t        TUPLE      (MARK at 0)
   10: p    PUT        1
```

```
   13: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
('a',)
result: ('a',)
```

We prefer to produce tuple literals, as above; but if the tuple is recursive, we need a more complicated construction. It used to be the case that the cPickle unpickler couldn't handle this case, but that's no longer true (see http://bugs.python.org/issue5794):

```
sage: v = ([],)
sage: v[0].append(v)
sage: test_pickle(pickle.dumps(v))
    0: (    MARK
    1: (        MARK
    2: l            LIST       (MARK at 1)
    3: p        PUT        0
    6: (        MARK
    7: g            GET        0
   10: t            TUPLE      (MARK at 6)
   11: p        PUT        1
   14: a        APPEND
   15: 0        POP
   16: 0        POP        (MARK at 0)
   17: g    GET        1
   20: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
si1 = []
si2 = (si1,)
list.append(si1, si2)
si2
result: ([(...)],)
```

TUPLE1()

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(('a',))
    0: \x80 PROTO      2
    2: U    SHORT_BINSTRING 'a'
    5: \x85 TUPLE1
    6: q    BINPUT     1
    8: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
('a',)
result: ('a',)
```

TUPLE2()

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(('a','b'))
    0: \x80 PROTO      2
    2: U    SHORT_BINSTRING 'a'
    5: U    SHORT_BINSTRING 'b'
    8: \x86 TUPLE2
    9: q    BINPUT     1
```

```
   11: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
('a', 'b')
result: ('a', 'b')
```

TUPLE3()

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(('a','b','c'))
    0: \x80 PROTO      2
    2: U    SHORT_BINSTRING 'a'
    5: U    SHORT_BINSTRING 'b'
    8: U    SHORT_BINSTRING 'c'
   11: \x87 TUPLE3
   12: q    BINPUT     1
   14: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
('a', 'b', 'c')
result: ('a', 'b', 'c')
```

UNICODE(*s*)

TESTS:

```
sage: import pickle
sage: from sage.misc.explain_pickle import *
sage: test_pickle(pickle.dumps(u'hi\u1234\U00012345'))
    0: V    UNICODE    u'hi\u1234\U00012345'
   20: p    PUT        0
   23: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
u'hi\u1234\U00012345'
result: u'hi\u1234\U00012345'
```

check_value(*v*)

Check that the given value is either a SageInputExpression or a PickleObject. Used for internal sanity checking.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: pe.check_value(7)
Traceback (most recent call last):
...
AssertionError
sage: pe.check_value(sib(7))
```

is_mutable_pickle_object(*v*)

Test whether a PickleObject is mutable (has never been converted to a SageInputExpression).

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
```

```
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: v = PickleObject(1, sib(1))
sage: pe.is_mutable_pickle_object(v)
True
sage: sib(v)
{atomic:1}
sage: pe.is_mutable_pickle_object(v)
False
```

pop()

Pop a value from the virtual machine's stack, and return it.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: pe.push(sib(7))
sage: pe.pop()
{atomic:7}
```

pop_to_mark()

Pop all values down to the 'mark' from the virtual machine's stack, and return the values as a list.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: pe.push_mark()
sage: pe.push(sib(7))
sage: pe.push(sib('hello'))
sage: pe.pop_to_mark()
[{atomic:7}, {atomic:'hello'}]
```

push(*v*)

Push a value onto the virtual machine's stack.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: pe.push(sib(7))
sage: pe.stack[-1]
{atomic:7}
```

push_and_share(*v*)

Push a value onto the virtual machine's stack; also mark it as shared for sage_input if we are in pedantic mode.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
```

```
sage: pe.push_and_share(sib(7))
sage: pe.stack[-1]
{atomic:7}
sage: pe.stack[-1]._sie_share
True
```

push_mark()

Push a 'mark' onto the virtual machine's stack.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: pe.push_mark()
sage: pe.stack[-1]
'mark'
sage: pe.stack[-1] is the_mark
True
```

run_pickle(*p*)

Given an (uncompressed) pickle as a string, run the pickle in this virtual machine. Once a STOP has been executed, return the result (a SageInputExpression representing code which, when evaluated, will give the value of the pickle).

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: sib(pe.run_pickle('T\5\0\0\0hello.'))
{atomic:'hello'}
```

share(*v*)

Mark a sage_input value as shared, if we are in pedantic mode.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: v = sib(7)
sage: v._sie_share
False
sage: pe.share(v)
{atomic:7}
sage: v._sie_share
True
```

class sage.misc.explain_pickle.**PickleInstance**(*klass*)

Bases: object

An object which can be used as the value of a PickleObject. Unlike other possible values of a PickleObject, a PickleInstance doesn't represent an exact value; instead, it gives the class (type) of the object.

class sage.misc.explain_pickle.**PickleObject**(*value, expression*)

Bases: object

Pickles have a stack-based virtual machine. The explain_pickle pickle interpreter mostly uses SageInputExpressions, from sage_input, as the stack values. However, sometimes we want some more information about the value on the stack, so that we can generate better (prettier, less confusing) code. In such cases, we push a PickleObject instead of a SageInputExpression. A PickleObject contains a value (which may be a standard Python value, or a PickleDict or PickleInstance), an expression (a SageInputExpression), and an "immutable" flag (which checks whether this object has been converted to a SageInputExpression; if it has, then we must not mutate the object, since the SageInputExpression would not reflect the changes).

class `sage.misc.explain_pickle.`**`TestAppendList`**

Bases: `list`

A subclass of list, with deliberately-broken append and extend methods. Used for testing explain_pickle.

`append``()`

A deliberately broken append method.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: v = TestAppendList()
sage: v.append(7)
Traceback (most recent call last):
...
TypeError: append() takes exactly 1 argument (2 given)
```

We can still append by directly using the list method: sage: list.append(v, 7) sage: v [7]

`extend``()`

A deliberately broken extend method.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: v = TestAppendList()
sage: v.extend([3,1,4,1,5,9])
Traceback (most recent call last):
...
TypeError: extend() takes exactly 1 argument (2 given)
```

We can still extend by directly using the list method: sage: list.extend(v, (3,1,4,1,5,9)) sage: v [3, 1, 4, 1, 5, 9]

class `sage.misc.explain_pickle.`**`TestAppendNonlist`**

Bases: `object`

A list-like class, carefully designed to test exact unpickling behavior. Used for testing explain_pickle.

class `sage.misc.explain_pickle.`**`TestBuild`**

Bases: `object`

A simple class with a __getstate__ but no __setstate__. Used for testing explain_pickle.

class `sage.misc.explain_pickle.`**`TestBuildSetstate`**

Bases: `sage.misc.explain_pickle.TestBuild`

A simple class with a __getstate__ and a __setstate__. Used for testing explain_pickle.

class `sage.misc.explain_pickle.`**`TestGlobalFunnyName`**

Bases: `object`

A featureless new-style class which has a name that's not a legal Python identifier.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: globals()['funny$name'] = TestGlobalFunnyName # see comment at end of file
sage: TestGlobalFunnyName.__name__
'funny$name'
sage: globals()['funny$name'] is TestGlobalFunnyName
True
```

class sage.misc.explain_pickle.**TestGlobalNewName**

Bases: object

A featureless new-style class. When you try to unpickle an instance of TestGlobalOldName, it is redirected to create an instance of this class instead. Used for testing explain_pickle.

EXAMPLES: sage: from sage.misc.explain_pickle import * sage: loads(dumps(TestGlobalOldName())) TestGlobalNewName

class sage.misc.explain_pickle.**TestGlobalOldName**

Bases: object

A featureless new-style class. When you try to unpickle an instance of this class, it is redirected to create a TestGlobalNewName instead. Used for testing explain_pickle.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: loads(dumps(TestGlobalOldName()))
TestGlobalNewName
```

class sage.misc.explain_pickle.**TestReduceGetinitargs**

An old-style class with a __getinitargs__ method. Used for testing explain_pickle.

class sage.misc.explain_pickle.**TestReduceNoGetinitargs**

An old-style class with no __getinitargs__ method. Used for testing explain_pickle.

sage.misc.explain_pickle.**explain_pickle**(*pickle=None*, *file=None*, *compress=True*, ***kwargs*)

Explain a pickle. That is, produce source code such that evaluating the code is equivalent to loading the pickle. Feeding the result of `explain_pickle` to `sage_eval` should be totally equivalent to loading the `pickle` with `cPickle`.

INPUTS:

- `pickle` – the pickle to explain, as a string (default: None)
- `file` – a filename of a pickle (default: None)
- **compress – if False, don't attempt to decompress the pickle** (default: True)
- **in_current_sage – if True, produce potentially simpler code that is** tied to the current version of Sage. (default: False)
- **default_assumptions – if True, produce potentially simpler code that** assumes that generic unpickling code will be used. This code may not actually work. (default: False)
- **eval – if True, then evaluate the resulting code and return the** evaluated result. (default: False)
- **preparse – if True, then produce code to be evaluated with** Sage's preparser; if False, then produce standard Python code; if None, then produce code that will work either with or without the preparser. (default: True)
- **pedantic – if True, then carefully ensures that the result has** at least as much sharing as the result of cPickle (it may have more, for immutable objects). (default: False)

Exactly one of `pickle` (a string containing a pickle) or `file` (the filename of a pickle) must be provided.

EXAMPLES:

```
sage: explain_pickle(dumps({('a', 'b'): [1r, 2r]}))
{('a', 'b'):[1r, 2r]}
sage: explain_pickle(dumps(RR(pi)), in_current_sage=True)
from sage.rings.real_mpfr import __create__RealNumber_version0
from sage.rings.real_mpfr import __create__RealField_version0
__create__RealNumber_version0(__create__RealField_version0(53r, False, 'RNDN'), '3.4gvml245kc0@0
sage: s = 'hi'
sage: explain_pickle(dumps((s, s)))
('hi', 'hi')
sage: explain_pickle(dumps((s, s)), pedantic=True)
si = 'hi'
(si, si)
sage: explain_pickle(dumps(5r))
5r
sage: explain_pickle(dumps(5r), preparse=False)
5
sage: explain_pickle(dumps(5r), preparse=None)
int(5)
sage: explain_pickle(dumps(22/7))
pg_make_rational = unpickle_global('sage.rings.rational', 'make_rational')
pg_make_rational('m/7')
sage: explain_pickle(dumps(22/7), in_current_sage=True)
from sage.rings.rational import make_rational
make_rational('m/7')
sage: explain_pickle(dumps(22/7), default_assumptions=True)
from sage.rings.rational import make_rational
make_rational('m/7')
```

`sage.misc.explain_pickle.`**`explain_pickle_string`**(*pickle*, *in_current_sage=False*, *default_assumptions=False*, *eval=False*, *preparse=True*, *pedantic=False*)

This is a helper function for explain_pickle. It takes a decompressed pickle string as input; other than that, its options are all the same as explain_pickle.

EXAMPLES:

```
sage: sage.misc.explain_pickle.explain_pickle_string(dumps("Hello, world", compress=False))
'Hello, world'
```

(See the documentation for `explain_pickle` for many more examples.)

`sage.misc.explain_pickle.`**`name_is_valid`**(*name*)

Test whether a string is a valid Python identifier. (We use a conservative test, that only allows ASCII identifiers.)

EXAMPLES:

```
sage: from sage.misc.explain_pickle import name_is_valid
sage: name_is_valid('fred')
True
sage: name_is_valid('Yes!ValidName')
False
sage: name_is_valid('_happy_1234')
True
```

`sage.misc.explain_pickle.`**`test_pickle`**(*p*, *verbose_eval=False*, *pedantic=False*, *args=()*)

Tests explain_pickle on a given pickle p. p can be:

- a string containing an uncompressed pickle (which will always end with a '.')
- a string containing a pickle fragment (not ending with '.') test_pickle will synthesize a pickle that will push args onto the stack (using persistent IDs), run the pickle fragment, and then STOP (if the string 'mark' occurs in args, then a mark will be pushed)
- an arbitrary object; test_pickle will pickle the object

Once it has a pickle, test_pickle will print the pickle's disassembly, run explain_pickle with in_current_sage=True and False, print the results, evaluate the results, unpickle the object with cPickle, and compare all three results.

If verbose_eval is True, then test_pickle will print messages before evaluating the pickles; this is to allow for tests where the unpickling prints messages (to verify that the same operations occur in all cases).

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(['a'])
    0: \x80 PROTO      2
    2: ]    EMPTY_LIST
    3: q    BINPUT     1
    5: U    SHORT_BINSTRING 'a'
    8: a    APPEND
    9: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
['a']
result: ['a']
```

sage.misc.explain_pickle.**unpickle_appends**(*lst*, *vals*)

Given a list (or list-like object) and a sequence of values, appends the values to the end of the list. This is careful to do so using the exact same technique that cPickle would use. Used by `explain_pickle`.

EXAMPLES:

```
sage: v = []
sage: unpickle_appends(v, (1, 2, 3))
sage: v
[1, 2, 3]
```

sage.misc.explain_pickle.**unpickle_build**(*obj*, *state*)

Set the state of an object. Used by `explain_pickle`.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: v = EmptyNewstyleClass()
sage: unpickle_build(v, {'hello': 42})
sage: v.hello
42
```

sage.misc.explain_pickle.**unpickle_extension**(*code*)

Takes an integer index and returns the extension object with that index. Used by `explain_pickle`.

EXAMPLES:

```
sage: from copy_reg import *
sage: add_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 42)
sage: unpickle_extension(42)
<class 'sage.misc.explain_pickle.EmptyNewstyleClass'>
sage: remove_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 42)
```

`sage.misc.explain_pickle.`**`unpickle_instantiate`**(*fn*, *args*)

Instantiate a new object of class fn with arguments args. Almost always equivalent to `fn(*args)`. Used by `explain_pickle`.

EXAMPLES:

```
sage: unpickle_instantiate(Integer, ('42',))
42
```

`sage.misc.explain_pickle.`**`unpickle_newobj`**(*klass*, *args*)

Create a new object; this corresponds to the C code klass->tp_new(klass, args, NULL). Used by `explain_pickle`.

EXAMPLES: sage: unpickle_newobj(tuple, ([1, 2, 3],)) (1, 2, 3)

`sage.misc.explain_pickle.`**`unpickle_persistent`**(*s*)

Takes an integer index and returns the persistent object with that index; works by calling whatever callable is stored in unpickle_persistent_loader. Used by `explain_pickle`.

EXAMPLES:

```
sage: import sage.misc.explain_pickle
sage: sage.misc.explain_pickle.unpickle_persistent_loader = lambda n: n+7
sage: unpickle_persistent(35)
42
```

CHAPTER

NINETEEN

GET RESOURCE USAGE OF PROCESS

AUTHORS:

- William Stein (2006-03-04): initial version

`sage.misc.getusage.`**`VmB`**`(`*VmKey*`)`

Function used internally by this module.

`sage.misc.getusage.`**`get_memory_usage`**`(`*t=None*`)`

Return memory usage.

INPUT:

- `t` - a float (default: None); output of an earlier call

OUTPUT:

- `Linux` - Returns float number (in megabytes)
- `OS X` - Returns float number (in megabytes) that matches VSIZE column of `top`
- `Solaris or OpenSolaris` - Returns float number (in megabytes) that matches RSS column of `prstat`. Depending on the memory usage, `prstat` will output the data in KB, MB or GB. In each case, the value returned by this function will always be in MB.
- `FreeBSD` - Returns float number (in megabytes) that matches RSS column of `ps -auxwww`
- `other` - not implemented for any other operating systems

EXAMPLES:

```
sage: t = get_memory_usage(); t  # random
873.98046875
```

Note:

- Currently, `get_memory_usage()` calls `prstat` on Solaris and OpenSolaris to get the data it requires. In the long term, a better solution would be to use Solaris system calls.
- In some instances, `top` may be used on OS X. This may break if the memory usage is greater than 9999 MB. However, normally `top` is not used on OS X.

`sage.misc.getusage.`**`linux_memory_usage`**`()`

Return memory usage in megabytes.

`sage.misc.getusage.`**`top`**`()`

Return the 'top' or 'prstat' line that contains this running Sage process. For FreeBSD, return the line containing this running Sage process from 'ps -axwww -o pid,user,vsz,rss,state,pri,nice,time,cpu,comm'.

OUTPUT:

•a string

EXAMPLES:

```
sage: top()              # random output
'72373 python       0.0%  0:01.36   1    14+  1197   39M+   34M+   55M+  130M+'
```

NOTES:

The external command 'top' (http://www.unixtop.org/) is called on Linux, and most other operating systems. The output format of 'top' is not consistent across all platforms and all versions of 'top'. If the `top()` function does not work in Sage, you may need to install 'top'.

The external command 'prstat' is called on the Solaris and OpenSolaris systems. That is part of Solaris, and will not need to be installed. The columns used in the 'prstat' output are:

```
PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
```

CHAPTER

TWENTY

MULTIDIMENSIONAL ENUMERATION

AUTHORS:

- Joel B. Mohler (2006-10-12)
- William Stein (2006-07-19)
- Jon Hanke

sage.misc.mrange.**cartesian_product_iterator**(*X*)

Iterate over the Cartesian product.

INPUT:

- `X` - list or tuple of lists

OUTPUT: iterator over the cartesian product of the elements of X

EXAMPLES:

```
sage: list(cartesian_product_iterator([[1,2], ['a','b']]))
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
sage: list(cartesian_product_iterator([]))
[()]
```

sage.misc.mrange.**mrange**(*sizes*, *typ=<type 'list'>*)

Return the multirange list with given sizes and type.

This is the list version of xmrange. Use xmrange for the iterator.

More precisely, return the iterator over all objects of type typ of n-tuples of Python ints with entries between 0 and the integers in the sizes list. The iterator is empty if sizes is empty or contains any non-positive integer.

INPUT:

- `sizes` - a list of nonnegative integers
- `typ` - (default: list) a type or class; more generally, something that can be called with a list as input.

OUTPUT: a list

EXAMPLES:

```
sage: mrange([3,2])
[[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]
sage: mrange([3,2], tuple)
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
sage: mrange([3,2], sum)
[0, 1, 1, 2, 2, 3]
```

Examples that illustrate empty multi-ranges:

```
sage: mrange([5,3,-2])
[]
sage: mrange([5,3,0])
[]
```

This example isn't empty, and shouldn't be. See trac #6561.

```
sage: mrange([])
[[]]
```

AUTHORS:

- Jon Hanke
- William Stein

`sage.misc.mrange.mrange_iter`(*iter_list*, *typ=<type 'list'>*)

Return the multirange list derived from the given list of iterators.

This is the list version of xmrange_iter. Use xmrange_iter for the iterator.

More precisely, return the iterator over all objects of type typ of n-tuples of Python ints with entries between 0 and the integers in the sizes list. The iterator is empty if sizes is empty or contains any non-positive integer.

INPUT:

- `iter_list` - a finite iterable of finite iterables
- `typ` - (default: list) a type or class; more generally, something that can be called with a list as input.

OUTPUT: a list

EXAMPLES:

```
sage: mrange_iter([range(3),[0,2]])
[[0, 0], [0, 2], [1, 0], [1, 2], [2, 0], [2, 2]]
sage: mrange_iter([['Monty','Flying'],['Python','Circus']], tuple)
[('Monty', 'Python'), ('Monty', 'Circus'), ('Flying', 'Python'), ('Flying', 'Circus')]
sage: mrange_iter([[2,3,5,7],[1,2]], sum)
[3, 4, 4, 5, 6, 7, 8, 9]
```

Examples that illustrate empty multi-ranges:

```
sage: mrange_iter([range(5),xrange(3),xrange(-2)])
[]
sage: mrange_iter([range(5),range(3),range(0)])
[]
```

This example isn't empty, and shouldn't be. See trac #6561.

```
sage: mrange_iter([])
[[]]
```

AUTHORS:

- Joel B. Mohler

class `sage.misc.mrange.xmrange`(*sizes*, *typ=<type 'list'>*)

Return the multirange iterate with given sizes and type.

More precisely, return the iterator over all objects of type typ of n-tuples of Python ints with entries between 0 and the integers in the sizes list. The iterator is empty if sizes is empty or contains any non-positive integer.

Use mrange for the non-iterator form.

INPUT:

- `sizes` - a list of nonnegative integers
- `typ` - (default: list) a type or class; more generally, something that can be called with a list as input.

OUTPUT: a generator

EXAMPLES: We create multi-range iterators, print them and also iterate through a tuple version.

```
sage: z = xmrange([3,2]);z
xmrange([3, 2])
sage: z = xmrange([3,2], tuple);z
xmrange([3, 2], <type 'tuple'>)
sage: for a in z:
...     print a
(0, 0)
(0, 1)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
```

We illustrate a few more iterations.

```
sage: list(xmrange([3,2]))
[[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]
sage: list(xmrange([3,2], tuple))
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
```

Here we compute the sum of each element of the multi-range iterator:

```
sage: list(xmrange([3,2], sum))
[0, 1, 1, 2, 2, 3]
```

Next we compute the product:

```
sage: list(xmrange([3,2], prod))
[0, 0, 0, 1, 0, 2]
```

Examples that illustrate empty multi-ranges.

```
sage: list(xmrange([5,3,-2]))
[]
sage: list(xmrange([5,3,0]))
[]
```

This example isn't empty, and shouldn't be. See trac #6561.

```
sage: list(xmrange([]))
[[]]
```

We use a multi-range iterator to iterate through the Cartesian product of sets.

```
sage: X = ['red', 'apple', 389]
sage: Y = ['orange', 'horse']
sage: for i,j in xmrange([len(X), len(Y)]):
...     print (X[i], Y[j])
('red', 'orange')
('red', 'horse')
('apple', 'orange')
('apple', 'horse')
```

```
(389, 'orange')
(389, 'horse')
```

AUTHORS:

- Jon Hanke
- William Stein

class `sage.misc.mrange.`**`xmrange_iter`**(*iter_list*, *typ=<type 'list'>*)

Return the multirange iterate derived from the given iterators and type.

Note: This basically gives you the Cartesian product of sets.

More precisely, return the iterator over all objects of type typ of n-tuples of Python ints with entries between 0 and the integers in the sizes list. The iterator is empty if sizes is empty or contains any non-positive integer.

Use mrange_iter for the non-iterator form.

INPUT:

- **`iter_list` - a list of objects usable as iterators (possibly** lists)
- **`typ` - (default: list) a type or class; more generally,** something that can be called with a list as input.

OUTPUT: a generator

EXAMPLES: We create multi-range iterators, print them and also iterate through a tuple version.

```
sage: z = xmrange_iter([xrange(3),xrange(2)]);z
xmrange_iter([xrange(3), xrange(2)])
sage: z = xmrange_iter([range(3),range(2)], tuple);z
xmrange_iter([[0, 1, 2], [0, 1]], <type 'tuple'>)
sage: for a in z:
...     print a
(0, 0)
(0, 1)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
```

We illustrate a few more iterations.

```
sage: list(xmrange_iter([range(3),range(2)]))
[[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]
sage: list(xmrange_iter([range(3),range(2)], tuple))
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
```

Here we compute the sum of each element of the multi-range iterator:

```
sage: list(xmrange_iter([range(3),range(2)], sum))
[0, 1, 1, 2, 2, 3]
```

Next we compute the product:

```
sage: list(xmrange_iter([range(3),range(2)], prod))
[0, 0, 0, 1, 0, 2]
```

Examples that illustrate empty multi-ranges.

```
sage: list(xmrange_iter([xrange(5),xrange(3),xrange(-2)]))
[]
sage: list(xmrange_iter([xrange(5),xrange(3),xrange(0)]))
[]
```

This example isn't empty, and shouldn't be. See trac #6561.

```
sage: list(xmrange_iter([]))
[[]]
```

We use a multi-range iterator to iterate through the Cartesian product of sets.

```
sage: X = ['red', 'apple', 389]
sage: Y = ['orange', 'horse']
sage: for i,j in xmrange_iter([X, Y], tuple):
...     print (i, j)
('red', 'orange')
('red', 'horse')
('apple', 'orange')
('apple', 'horse')
(389, 'orange')
(389, 'horse')
```

AUTHORS:

- Joel B. Mohler

cardinality()

Return the cardinality of this iterator.

EXAMPLES:

```
sage: C = cartesian_product_iterator([xrange(3), xrange(4)])
sage: C.cardinality()
12
sage: C = cartesian_product_iterator([ZZ,QQ])
sage: C.cardinality()
+Infinity
sage: C = cartesian_product_iterator([ZZ,[]])
sage: C.cardinality()
0
```

CHAPTER TWENTYONE

INSTALLING SHORTCUT SCRIPTS

`sage.misc.dist.`**`install_scripts`**`(`*directory=None*, *ignore_existing=False*`)`

Running `install_scripts(directory)` creates scripts in the given directory that run various software components included with Sage. Each of these scripts essentially just runs `sage --CMD` where `CMD` is also the name of the script:

- 'gap' runs GAP
- 'gp' runs the PARI/GP interpreter
- 'hg' runs Mercurial
- 'ipython' runs IPython
- 'maxima' runs Maxima
- 'mwrank' runs mwrank
- 'R' runs R
- 'singular' runs Singular
- 'sqlite3' runs SQLite version 3
- 'kash' runs Kash if it is installed (Kash is an optional Sage package)
- 'M2' runs Macaulay2 if it is installed (Macaulay2 is an experimental Sage package)

This command:

- verbosely tells you which scripts it adds, and
- will *not* overwrite any scripts you already have in the given directory.

INPUT:

- `directory` - string; the directory into which to put the scripts. This directory must exist and the user must have write and execute permissions.
- `ignore_existing` - bool (optional, default False): if True, install script even if another version of the program is in your path.

OUTPUT: Verbosely prints what it is doing and creates files in `directory` that are world executable and readable.

Note: You may need to run `sage` as `root` in order to run `install_scripts` successfully, since the user running `sage` needs write permissions on `directory`. Note that one good candidate for `directory` is `'/usr/local/bin'`, so from the shell prompt, you could run

```
sudo sage -c "install_scripts('/usr/local/bin')"
```

Note: Running `install_scripts(directory)` will be most helpful if `directory` is in your path.

AUTHORS:

- William Stein: code / design
- Arthur Gaer: design
- John Palmieri: revision, 2011-07 (trac ticket #11602)

EXAMPLES:

```
sage: install_scripts(str(SAGE_TMP), ignore_existing=True)
Checking that Sage has the command 'gap' installed
...
```

CHAPTER

TWENTYTWO

FUNCTIONAL NOTATION

These are functions so that you can write foo(x) instead of x.foo() in certain common cases.

AUTHORS:

- William Stein: Initial version
- David Joyner (2005-12-20): More Examples

sage.misc.functional.**N**(*x*, *prec=None*, *digits=None*, *algorithm=None*)

Returns a numerical approximation of an object x with at least prec bits (or decimal digits) of precision.

Note: Both upper case N and lower case n are aliases for numerical_approx(), and all three may be used as methods.

INPUT:

- x - an object that has a numerical_approx method, or can be coerced into a real or complex field
- prec (optional) - an integer (bits of precision)
- digits (optional) - an integer (digits of precision)
- algorithm (optional) - a string specifying the algorithm to use for functions that implement more than one

If neither the prec or digits are specified, the default is 53 bits of precision. If both are specified, then prec is used.

EXAMPLES:

```
sage: numerical_approx(pi, 10)
3.1
sage: numerical_approx(pi, digits=10)
3.141592654
sage: numerical_approx(pi^2 + e, digits=20)
12.587886229548403854
sage: n(pi^2 + e)
12.5878862295484
sage: N(pi^2 + e)
12.5878862295484
sage: n(pi^2 + e, digits=50)
12.587886229548403854194778471228813633070946500941
sage: a = CC(-5).n(prec=40)
sage: b = ComplexField(40)(-5)
sage: a == b
True
sage: parent(a) is parent(b)
True
```

```
sage: numerical_approx(9)
9.00000000000000
```

You can also usually use method notation.

```
sage: (pi^2 + e).n()
12.5878862295484
sage: (pi^2 + e).N()
12.5878862295484
sage: (pi^2 + e).numerical_approx()
12.5878862295484
```

Vectors and matrices may also have their entries approximated.

```
sage: v = vector(RDF, [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)

sage: v = vector(CDF, [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: _.parent()
Vector space of dimension 3 over Complex Field with 53 bits of precision
sage: v.n(prec=20)
(1.0000, 2.0000, 3.0000)

sage: u = vector(QQ, [1/2, 1/3, 1/4])
sage: n(u, prec=15)
(0.5000, 0.3333, 0.2500)
sage: n(u, digits=5)
(0.50000, 0.33333, 0.25000)

sage: v = vector(QQ, [1/2, 0, 0, 1/3, 0, 0, 0, 1/4], sparse=True)
sage: u = v.numerical_approx(digits=4)
sage: u.is_sparse()
True
sage: u
(0.5000, 0.0000, 0.0000, 0.3333, 0.0000, 0.0000, 0.0000, 0.2500)

sage: A = matrix(QQ, 2, 3, range(6))
sage: A.n()
[0.000000000000000  1.00000000000000  2.00000000000000]
[ 3.00000000000000  4.00000000000000  5.00000000000000]

sage: B = matrix(Integers(12), 3, 8, srange(24))
sage: N(B, digits=2)
[0.00  1.0  2.0  3.0  4.0  5.0  6.0  7.0]
[ 8.0  9.0  10.  11. 0.00  1.0  2.0  3.0]
[ 4.0  5.0  6.0  7.0  8.0  9.0  10.  11.]
```

Internally, numerical approximations of real numbers are stored in base-2. Therefore, numbers which look the same in their decimal expansion might be different:

```
sage: x=N(pi, digits=3); x
3.14
sage: y=N(3.14, digits=3); y
3.14
sage: x==y
False
```

```
sage: x.str(base=2)
'11.001001000100'
sage: y.str(base=2)
'11.001000111101'
```

Increasing the precision of a floating point number is not allowed:

```
sage: CC(-5).n(prec=100)
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 100 bits, use at most 53 bits
sage: n(1.3r, digits=20)
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 70 bits, use at most 53 bits
sage: RealField(24).pi().n()
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 53 bits, use at most 24 bits
```

As an exceptional case, `digits=1` usually leads to 2 digits (one significant) in the decimal output (see trac ticket #11647):

```
sage: N(pi, digits=1)
3.2
sage: N(pi, digits=2)
3.1
sage: N(100*pi, digits=1)
320.
sage: N(100*pi, digits=2)
310.
```

In the following example, `pi` and `3` are both approximated to two bits of precision and then subtracted, which kills two bits of precision:

```
sage: N(pi, prec=2)
3.0
sage: N(3, prec=2)
3.0
sage: N(pi - 3, prec=2)
0.00
```

TESTS:

```
sage: numerical_approx(I)
1.00000000000000*I
sage: x = QQ['x'].gen()
sage: F.<k> = NumberField(x^2+2, embedding=sqrt(CC(2))*CC.0)
sage: numerical_approx(k)
1.41421356237309*I

sage: type(numerical_approx(CC(1/2)))
<type 'sage.rings.complex_number.ComplexNumber'>
```

The following tests trac ticket #10761, in which `n()` would break when called on complex-valued algebraic numbers.

```
sage: E = matrix(3, [3,1,6,5,2,9,7,3,13]).eigenvalues(); E
[18.16815365088822?, -0.08407682544410650? - 0.2190261484802906?*I, -0.08407682544410650? + 0.21
sage: E[1].parent()
```

```
Algebraic Field
sage: [a.n() for a in E]
[18.1681536508882, -0.0840768254441065 - 0.219026148480291*I, -0.0840768254441065 + 0.2190261484
```

Make sure we've rounded up log(10,2) enough to guarantee sufficient precision (trac ticket #10164):

```
sage: ks = 4*10**5, 10**6
sage: check_str_length = lambda k: len(str(numerical_approx(1+10**-k,digits=k+1)))-1 >= k+1
sage: check_precision = lambda k: numerical_approx(1+10**-k,digits=k+1)-1 > 0
sage: all(check_str_length(k) and check_precision(k) for k in ks)
True
```

Testing we have sufficient precision for the golden ratio (trac ticket #12163), note that the decimal point adds 1 to the string length:

```
sage: len(str(n(golden_ratio, digits=5000)))
5001
sage: len(str(n(golden_ratio, digits=5000000)))  # long time (4s on sage.math, 2012)
5000001
```

Check that trac ticket #14778 is fixed:

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

sage.misc.functional.**additive_order**(*x*)

Returns the additive order of x.

EXAMPLES:

```
sage: additive_order(5)
+Infinity
sage: additive_order(Mod(5,11))
11
sage: additive_order(Mod(4,12))
3
```

sage.misc.functional.**base_field**(*x*)

Returns the base field over which x is defined.

EXAMPLES:

```
sage: R = PolynomialRing(GF(7), 'x')
sage: base_ring(R)
Finite Field of size 7
sage: base_field(R)
Finite Field of size 7
```

This catches base rings which are fields as well, but does not implement a `base_field` method for objects which do not have one:

```
sage: R.base_field()
Traceback (most recent call last):
...
AttributeError: 'PolynomialRing_dense_mod_p_with_category' object has no attribute 'base_field'
```

sage.misc.functional.**base_ring**(*x*)

Returns the base ring over which x is defined.

EXAMPLES:

```
sage: R = PolynomialRing(GF(7), 'x')
sage: base_ring(R)
Finite Field of size 7
```

`sage.misc.functional.`**`basis`**(*x*)

Returns the fixed basis of x.

EXAMPLES:

```
sage: V = VectorSpace(QQ,3)
sage: S = V.subspace([[1,2,0],[2,2,-1]])
sage: basis(S)
[
(1, 0, -1),
(0, 1, 1/2)
]
```

`sage.misc.functional.`**`category`**(*x*)

Returns the category of x.

EXAMPLES:

```
sage: V = VectorSpace(QQ,3)
sage: category(V)
Category of vector spaces over Rational Field
```

`sage.misc.functional.`**`characteristic_polynomial`**(*x*, *var='x'*)

Returns the characteristic polynomial of x in the given variable.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: charpoly(A)
x^3 - 15*x^2 - 18*x
sage: charpoly(A, 't')
t^3 - 15*t^2 - 18*t
```

```
sage: k.<alpha> = GF(7^10); k
Finite Field in alpha of size 7^10
sage: alpha.charpoly('T')
T^10 + T^6 + T^5 + 4*T^4 + T^3 + 2*T^2 + 3*T + 3
sage: characteristic_polynomial(alpha, 'T')
T^10 + T^6 + T^5 + 4*T^4 + T^3 + 2*T^2 + 3*T + 3
```

Ensure the variable name of the polynomial does not conflict with variables used within the matrix, and that non-integral powers of variables don't confuse the computation (trac ticket #14403):

```
sage: y = var('y')
sage: a = matrix([[x,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
sage: characteristic_polynomial(a).list()
[x, -3*x - 1, 3*x + 3, -x - 3, 1]
sage: b = matrix([[y^(1/2),0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
sage: charpoly(b).list()
[sqrt(y), -3*sqrt(y) - 1, 3*sqrt(y) + 3, -sqrt(y) - 3, 1]
```

`sage.misc.functional.`**`charpoly`**(*x*, *var='x'*)

Returns the characteristic polynomial of x in the given variable.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: charpoly(A)
x^3 - 15*x^2 - 18*x
sage: charpoly(A, 't')
t^3 - 15*t^2 - 18*t

sage: k.<alpha> = GF(7^10); k
Finite Field in alpha of size 7^10
sage: alpha.charpoly('T')
T^10 + T^6 + T^5 + 4*T^4 + T^3 + 2*T^2 + 3*T + 3
sage: characteristic_polynomial(alpha, 'T')
T^10 + T^6 + T^5 + 4*T^4 + T^3 + 2*T^2 + 3*T + 3
```

Ensure the variable name of the polynomial does not conflict with variables used within the matrix, and that non-integral powers of variables don't confuse the computation (trac ticket #14403):

```
sage: y = var('y')
sage: a = matrix([[x,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
sage: characteristic_polynomial(a).list()
[x, -3*x - 1, 3*x + 3, -x - 3, 1]
sage: b = matrix([[y^(1/2),0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
sage: charpoly(b).list()
[sqrt(y), -3*sqrt(y) - 1, 3*sqrt(y) + 3, -sqrt(y) - 3, 1]
```

sage.misc.functional.**coerce**(*P*, *x*)

Attempts to coerce x to type P if possible.

EXAMPLES:

```
sage: type(5)
<type 'sage.rings.integer.Integer'>
sage: type(coerce(QQ,5))
<type 'sage.rings.rational.Rational'>
```

sage.misc.functional.**cyclotomic_polynomial**(*n*, *var='x'*)

Returns the n^{th} cyclotomic polynomial.

EXAMPLES:

```
sage: cyclotomic_polynomial(3)
x^2 + x + 1
sage: cyclotomic_polynomial(4)
x^2 + 1
sage: cyclotomic_polynomial(9)
x^6 + x^3 + 1
sage: cyclotomic_polynomial(10)
x^4 - x^3 + x^2 - x + 1
sage: cyclotomic_polynomial(11)
x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
```

sage.misc.functional.**decomposition**(*x*)

Returns the decomposition of x.

EXAMPLES:

```
sage: M = matrix([[2, 3], [3, 4]])
sage: M.decomposition()
[
(Ambient free module of rank 2 over the principal ideal domain Integer Ring, True)
]
```

```
sage: G.<a,b> = DirichletGroup(20)
sage: c = a*b
sage: d = c.decomposition(); d
[Dirichlet character modulo 4 of conductor 4 mapping 3 |--> -1,
Dirichlet character modulo 5 of conductor 5 mapping 2 |--> zeta4]
sage: d[0].parent()
Group of Dirichlet characters of modulus 4 over Cyclotomic Field of order 4 and degree 2
```

sage.misc.functional.**denominator**(*x*)

Returns the denominator of x.

EXAMPLES:

```
sage: denominator(17/11111)
11111
sage: R.<x> = PolynomialRing(QQ)
sage: F = FractionField(R)
sage: r = (x+1)/(x-1)
sage: denominator(r)
x - 1
```

sage.misc.functional.**det**(*x*)

Returns the determinant of x.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: det(A)
0
```

sage.misc.functional.**dim**(*x*)

Returns the dimension of x.

EXAMPLES:

```
sage: V = VectorSpace(QQ,3)
sage: S = V.subspace([[1,2,0],[2,2,-1]])
sage: dimension(S)
2
```

sage.misc.functional.**dimension**(*x*)

Returns the dimension of x.

EXAMPLES:

```
sage: V = VectorSpace(QQ,3)
sage: S = V.subspace([[1,2,0],[2,2,-1]])
sage: dimension(S)
2
```

sage.misc.functional.**disc**(*x*)

Returns the discriminant of x.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^29 - 17*x - 1, 'alpha')
sage: K = S.number_field()
sage: discriminant(K)
-15975100446626038280218213241591829458737190477345113376757479850566957249523
```

`sage.misc.functional.`**`discriminant`**`(x)`

Returns the discriminant of x.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^29 - 17*x - 1, 'alpha')
sage: K = S.number_field()
sage: discriminant(K)
-15975100446626038280218213241591829458737190477345113376757479850566957249523
```

`sage.misc.functional.`**`eta`**`(x)`

Returns the value of the eta function at x, which must be in the upper half plane.

The η function is

$$\eta(z) = e^{\pi iz/12} \prod_{n=1}^{\infty} (1 - e^{2\pi inz})$$

EXAMPLES:

```
sage: eta(1+I)
0.7420487758365647 + 0.1988313702299107*I
```

`sage.misc.functional.`**`fcp`**`(x, var='x')`

Returns the factorization of the characteristic polynomial of x.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: fcp(A, 'x')
x * (x^2 - 15*x - 18)
```

`sage.misc.functional.`**`gen`**`(x)`

Returns the generator of x.

EXAMPLES:

```
sage: R.<x> = QQ[]; R
Univariate Polynomial Ring in x over Rational Field
sage: gen(R)
x
sage: gen(GF(7))
1
sage: A = AbelianGroup(1, [23])
sage: gen(A)
f
```

`sage.misc.functional.`**`gens`**`(x)`

Returns the generators of x.

EXAMPLES:

```
sage: R.<x,y> = SR[]
sage: R
Multivariate Polynomial Ring in x, y over Symbolic Ring
sage: gens(R)
(x, y)
sage: A = AbelianGroup(5, [5,5,7,8,9])
```

```
sage: gens(A)
(f0, f1, f2, f3, f4)
```

sage.misc.functional.**hecke_operator**(*x*, *n*)

Returns the n-th Hecke operator T_n acting on x.

EXAMPLES:

```
sage: M = ModularSymbols(1,12)
sage: hecke_operator(M,5)
Hecke operator T_5 on Modular Symbols space of dimension 3 for Gamma_0(1) of weight 12 with sign
```

sage.misc.functional.**image**(*x*)

Returns the image of x.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: image(A)
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
```

sage.misc.functional.**integral**(*x*, **args*, ***kwds*)

Returns an indefinite or definite integral of an object x.

First call x.integral() and if that fails make an object and integrate it using Maxima, maple, etc, as specified by algorithm.

For symbolic expression calls `sage.calculus.calculus.integral()` - see this function for available options.

EXAMPLES:

```
sage: f = cyclotomic_polynomial(10)
sage: integral(f)
1/5*x^5 - 1/4*x^4 + 1/3*x^3 - 1/2*x^2 + x

sage: integral(sin(x),x)
-cos(x)

sage: y = var('y')
sage: integral(sin(x),y)
y*sin(x)

sage: integral(sin(x), x, 0, pi/2)
1
sage: sin(x).integral(x, 0,pi/2)
1
sage: integral(exp(-x), (x, 1, oo))
e^(-1)
```

Numerical approximation:

```
sage: h = integral(tan(x)/x, (x, 1, pi/3)); h
integrate(tan(x)/x, x, 1, 1/3*pi)
sage: h.n()
0.07571599101...
```

Specific algorithm can be used for integration:

```
sage: integral(sin(x)^2, x, algorithm='maxima')
1/2*x - 1/4*sin(2*x)
sage: integral(sin(x)^2, x, algorithm='sympy')
-1/2*cos(x)*sin(x) + 1/2*x
```

TESTS:

A symbolic integral from trac ticket #11445 that was incorrect in earlier versions of Maxima:

```
sage: f = abs(x - 1) + abs(x + 1) - 2*abs(x)
sage: integrate(f, (x, -Infinity, Infinity))
2
```

Another symbolic integral, from trac ticket #11238, that used to return zero incorrectly; with Maxima 5.26.0 one gets `1/2*sqrt(pi)*e^(1/4)`, whereas with 5.29.1, and even more so with 5.33.0, the expression is less pleasant, but still has the same value. Unfortunately, the computation takes a very long time with the default settings, so we temporarily use the Maxima setting `domain: real`:

```
sage: sage.calculus.calculus.maxima('domain: real')
real
sage: f = exp(-x) * sinh(sqrt(x))
sage: t = integrate(f, x, 0, Infinity); t            # long time
1/4*sqrt(pi)*(erf(1) - 1)*e^(1/4) - 1/4*(sqrt(pi)*(erf(1) - 1) - sqrt(pi) + 2*e^(-1) - 2)*e^(1/4
sage: t.canonicalize_radical()  # long time
1/2*sqrt(pi)*e^(1/4)
sage: sage.calculus.calculus.maxima('domain: complex')
complex
```

An integral which used to return -1 before maxima 5.28. See trac ticket #12842:

```
sage: f = e^(-2*x)/sqrt(1-e^(-2*x))
sage: integrate(f, x, 0, infinity)
1
```

This integral would cause a stack overflow in earlier versions of Maxima, crashing sage. See trac ticket #12377. We don't care about the result here, just that the computation completes successfully:

```
sage: y = (x^2)*exp(x) / (1 + exp(x))^2
sage: _ = integrate(y, x, -1000, 1000)
```

`sage.misc.functional.`**`integral_closure`**(*x*)

Returns the integral closure of x.

EXAMPLES:

```
sage: integral_closure(QQ)
Rational Field
sage: K.<a> = QuadraticField(5)
sage: O2 = K.order(2*a); O2
Order in Number Field in a with defining polynomial x^2 - 5
sage: integral_closure(O2)
Maximal Order in Number Field in a with defining polynomial x^2 - 5
```

`sage.misc.functional.`**`integrate`**(*x*, **args*, ***kwds*)

Returns an indefinite or definite integral of an object x.

First call x.integral() and if that fails make an object and integrate it using Maxima, maple, etc, as specified by algorithm.

For symbolic expression calls `sage.calculus.calculus.integral()` - see this function for available options.

EXAMPLES:

```
sage: f = cyclotomic_polynomial(10)
sage: integral(f)
1/5*x^5 - 1/4*x^4 + 1/3*x^3 - 1/2*x^2 + x

sage: integral(sin(x),x)
-cos(x)

sage: y = var('y')
sage: integral(sin(x),y)
y*sin(x)

sage: integral(sin(x), x, 0, pi/2)
1
sage: sin(x).integral(x, 0,pi/2)
1
sage: integral(exp(-x), (x, 1, oo))
e^(-1)
```

Numerical approximation:

```
sage: h = integral(tan(x)/x, (x, 1, pi/3)); h
integrate(tan(x)/x, x, 1, 1/3*pi)
sage: h.n()
0.07571599101...
```

Specific algorithm can be used for integration:

```
sage: integral(sin(x)^2, x, algorithm='maxima')
1/2*x - 1/4*sin(2*x)
sage: integral(sin(x)^2, x, algorithm='sympy')
-1/2*cos(x)*sin(x) + 1/2*x
```

TESTS:

A symbolic integral from trac ticket #11445 that was incorrect in earlier versions of Maxima:

```
sage: f = abs(x - 1) + abs(x + 1) - 2*abs(x)
sage: integrate(f, (x, -Infinity, Infinity))
2
```

Another symbolic integral, from trac ticket #11238, that used to return zero incorrectly; with Maxima 5.26.0 one gets `1/2*sqrt(pi)*e^(1/4)`, whereas with 5.29.1, and even more so with 5.33.0, the expression is less pleasant, but still has the same value. Unfortunately, the computation takes a very long time with the default settings, so we temporarily use the Maxima setting `domain: real`:

```
sage: sage.calculus.calculus.maxima('domain: real')
real
sage: f = exp(-x) * sinh(sqrt(x))
sage: t = integrate(f, x, 0, Infinity); t            # long time
1/4*sqrt(pi)*(erf(1) - 1)*e^(1/4) - 1/4*(sqrt(pi)*(erf(1) - 1) - sqrt(pi) + 2*e^(-1) - 2)*e^(1/4
sage: t.canonicalize_radical()  # long time
1/2*sqrt(pi)*e^(1/4)
sage: sage.calculus.calculus.maxima('domain: complex')
complex
```

An integral which used to return -1 before maxima 5.28. See trac ticket #12842:

```
sage: f = e^(-2*x)/sqrt(1-e^(-2*x))
sage: integrate(f, x, 0, infinity)
1
```

This integral would cause a stack overflow in earlier versions of Maxima, crashing sage. See trac ticket #12377. We don't care about the result here, just that the computation completes successfully:

```
sage: y = (x^2)*exp(x) / (1 + exp(x))^2
sage: _ = integrate(y, x, -1000, 1000)
```

`sage.misc.functional.`**`interval`**`(a, b)`

Integers between a and b *inclusive* (a and b integers).

EXAMPLES:

```
sage: I = interval(1,3)
sage: 2 in I
True
sage: 1 in I
True
sage: 4 in I
False
```

`sage.misc.functional.`**`is_commutative`**`(x)`

Returns whether or not x is commutative.

EXAMPLES:

```
sage: R = PolynomialRing(QQ, 'x')
sage: is_commutative(R)
True
```

`sage.misc.functional.`**`is_even`**`(x)`

Returns whether or not an integer x is even, e.g., divisible by 2.

EXAMPLES:

```
sage: is_even(-1)
False
sage: is_even(4)
True
sage: is_even(-2)
True
```

`sage.misc.functional.`**`is_field`**`(x)`

Returns whether or not x is a field.

EXAMPLES:

```
sage: R = PolynomialRing(QQ, 'x')
sage: F = FractionField(R)
sage: is_field(F)
True
```

`sage.misc.functional.`**`is_integrally_closed`**`(x)`

Returns whether x is integrally closed.

EXAMPLES:

```
sage: is_integrally_closed(QQ)
True
sage: K.<a> = NumberField(x^2 + 189*x + 394)
```

```
sage: R = K.order(2*a)
sage: is_integrally_closed(R)
False
```

`sage.misc.functional.is_odd(x)`

Returns whether or not x is odd. This is by definition the complement of is_even.

EXAMPLES:

```
sage: is_odd(-2)
False
sage: is_odd(-3)
True
sage: is_odd(0)
False
sage: is_odd(1)
True
```

`sage.misc.functional.isqrt(x)`

Returns an integer square root, i.e., the floor of a square root.

EXAMPLES:

```
sage: isqrt(10)
3
sage: isqrt(10r)
3
```

`sage.misc.functional.kernel(x)`

Returns the left kernel of x.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,2)
sage: A = M([1,2,3,4,5,6])
sage: kernel(A)
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
sage: kernel(A.transpose())
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

Here are two corner cases: sage: M=MatrixSpace(QQ,0,3) sage: A=M([]) sage: kernel(A) Vector space of degree 0 and dimension 0 over Rational Field Basis matrix: [] sage: kernel(A.transpose()).basis() [(1, 0, 0), (0, 1, 0), (0, 0, 1)]

`sage.misc.functional.krull_dimension(x)`

Returns the Krull dimension of x.

EXAMPLES:

```
sage: krull_dimension(QQ)
0
sage: krull_dimension(ZZ)
1
sage: krull_dimension(ZZ[sqrt(5)])
1
sage: U.<x,y,z> = PolynomialRing(ZZ,3); U
```

```
Multivariate Polynomial Ring in x, y, z over Integer Ring
sage: U.krull_dimension()
4
```

sage.misc.functional.**lift**(*x*)

Lift an object of a quotient ring R/I to R.

EXAMPLES: We lift an integer modulo 3.

```
sage: Mod(2,3).lift()
2
```

We lift an element of a quotient polynomial ring.

```
sage: R.<x> = QQ['x']
sage: S.<xmod> = R.quo(x^2 + 1)
sage: lift(xmod-7)
x - 7
```

sage.misc.functional.**log**(*x*, *b=None*)

Returns the log of x to the base b. The default base is e.

INPUT:

- x - number
- b - base (default: None, which means natural log)

OUTPUT: number

Note: In Magma, the order of arguments is reversed from in Sage, i.e., the base is given first. We use the opposite ordering, so the base can be viewed as an optional second argument.

EXAMPLES:

```
sage: log(e^2)
2
sage: log(16,2)
4
sage: log(3.)
1.09861228866811
```

sage.misc.functional.**minimal_polynomial**(*x*, *var='x'*)

Returns the minimal polynomial of x.

EXAMPLES:

```
sage: a = matrix(ZZ, 2, [1..4])
sage: minpoly(a)
x^2 - 5*x - 2
sage: minpoly(a,'t')
t^2 - 5*t - 2
sage: minimal_polynomial(a)
x^2 - 5*x - 2
sage: minimal_polynomial(a,'theta')
theta^2 - 5*theta - 2
```

sage.misc.functional.**minpoly**(*x*, *var='x'*)

Returns the minimal polynomial of x.

EXAMPLES:

```
sage: a = matrix(ZZ, 2, [1..4])
sage: minpoly(a)
x^2 - 5*x - 2
sage: minpoly(a,'t')
t^2 - 5*t - 2
sage: minimal_polynomial(a)
x^2 - 5*x - 2
sage: minimal_polynomial(a,'theta')
theta^2 - 5*theta - 2
```

sage.misc.functional.**multiplicative_order**(*x*)

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

EXAMPLES:

```
sage: a = mod(5,11)
sage: multiplicative_order(a)
5
sage: multiplicative_order(mod(2,11))
10
sage: multiplicative_order(mod(2,12))
Traceback (most recent call last):
...
ArithmeticError: multiplicative order of 2 not defined since it is not a unit modulo 12
```

sage.misc.functional.**n**(*x*, *prec=None*, *digits=None*, *algorithm=None*)

Returns a numerical approximation of an object `x` with at least `prec` bits (or decimal `digits`) of precision.

Note: Both upper case `N` and lower case `n` are aliases for `numerical_approx()`, and all three may be used as methods.

INPUT:

- `x` - an object that has a numerical_approx method, or can be coerced into a real or complex field
- `prec` (optional) - an integer (bits of precision)
- `digits` (optional) - an integer (digits of precision)
- `algorithm` (optional) - a string specifying the algorithm to use for functions that implement more than one

If neither the `prec` or `digits` are specified, the default is 53 bits of precision. If both are specified, then `prec` is used.

EXAMPLES:

```
sage: numerical_approx(pi, 10)
3.1
sage: numerical_approx(pi, digits=10)
3.141592654
sage: numerical_approx(pi^2 + e, digits=20)
12.587886229548403854
sage: n(pi^2 + e)
12.5878862295484
sage: N(pi^2 + e)
12.5878862295484
sage: n(pi^2 + e, digits=50)
12.587886229548403854194778471228813633070946500941
sage: a = CC(-5).n(prec=40)
```

```
sage: b = ComplexField(40)(-5)
sage: a == b
True
sage: parent(a) is parent(b)
True
sage: numerical_approx(9)
9.00000000000000
```

You can also usually use method notation.

```
sage: (pi^2 + e).n()
12.5878862295484
sage: (pi^2 + e).N()
12.5878862295484
sage: (pi^2 + e).numerical_approx()
12.5878862295484
```

Vectors and matrices may also have their entries approximated.

```
sage: v = vector(RDF, [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)

sage: v = vector(CDF, [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: _.parent()
Vector space of dimension 3 over Complex Field with 53 bits of precision
sage: v.n(prec=20)
(1.0000, 2.0000, 3.0000)

sage: u = vector(QQ, [1/2, 1/3, 1/4])
sage: n(u, prec=15)
(0.5000, 0.3333, 0.2500)
sage: n(u, digits=5)
(0.50000, 0.33333, 0.25000)

sage: v = vector(QQ, [1/2, 0, 0, 1/3, 0, 0, 0, 1/4], sparse=True)
sage: u = v.numerical_approx(digits=4)
sage: u.is_sparse()
True
sage: u
(0.5000, 0.0000, 0.0000, 0.3333, 0.0000, 0.0000, 0.0000, 0.2500)

sage: A = matrix(QQ, 2, 3, range(6))
sage: A.n()
[0.000000000000000  1.00000000000000  2.00000000000000]
[ 3.00000000000000  4.00000000000000  5.00000000000000]

sage: B = matrix(Integers(12), 3, 8, srange(24))
sage: N(B, digits=2)
[0.00  1.0  2.0  3.0  4.0  5.0  6.0  7.0]
[ 8.0  9.0  10.  11. 0.00  1.0  2.0  3.0]
[ 4.0  5.0  6.0  7.0  8.0  9.0  10.  11.]
```

Internally, numerical approximations of real numbers are stored in base-2. Therefore, numbers which look the same in their decimal expansion might be different:

```
sage: x=N(pi, digits=3); x
3.14
sage: y=N(3.14, digits=3); y
3.14
sage: x==y
False
sage: x.str(base=2)
'11.001001000100'
sage: y.str(base=2)
'11.001000111101'
```

Increasing the precision of a floating point number is not allowed:

```
sage: CC(-5).n(prec=100)
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 100 bits, use at most 53 bits
sage: n(1.3r, digits=20)
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 70 bits, use at most 53 bits
sage: RealField(24).pi().n()
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 53 bits, use at most 24 bits
```

As an exceptional case, `digits=1` usually leads to 2 digits (one significant) in the decimal output (see trac ticket #11647):

```
sage: N(pi, digits=1)
3.2
sage: N(pi, digits=2)
3.1
sage: N(100*pi, digits=1)
320.
sage: N(100*pi, digits=2)
310.
```

In the following example, `pi` and `3` are both approximated to two bits of precision and then subtracted, which kills two bits of precision:

```
sage: N(pi, prec=2)
3.0
sage: N(3, prec=2)
3.0
sage: N(pi - 3, prec=2)
0.00
```

TESTS:

```
sage: numerical_approx(I)
1.00000000000000*I
sage: x = QQ['x'].gen()
sage: F.<k> = NumberField(x^2+2, embedding=sqrt(CC(2))*CC.0)
sage: numerical_approx(k)
1.41421356237309*I

sage: type(numerical_approx(CC(1/2)))
<type 'sage.rings.complex_number.ComplexNumber'>
```

The following tests trac ticket #10761, in which `n()` would break when called on complex-valued algebraic numbers.

```
sage: E = matrix(3, [3,1,6,5,2,9,7,3,13]).eigenvalues(); E
[18.16815365088822?, -0.08407682544410650? - 0.2190261484802906?*I, -0.08407682544410650? + 0.21
sage: E[1].parent()
Algebraic Field
sage: [a.n() for a in E]
[18.1681536508882, -0.0840768254441065 - 0.219026148480291*I, -0.0840768254441065 + 0.2190261484
```

Make sure we've rounded up log(10,2) enough to guarantee sufficient precision (trac ticket #10164):

```
sage: ks = 4*10**5, 10**6
sage: check_str_length = lambda k: len(str(numerical_approx(1+10**-k,digits=k+1)))-1 >= k+1
sage: check_precision = lambda k: numerical_approx(1+10**-k,digits=k+1)-1 > 0
sage: all(check_str_length(k) and check_precision(k) for k in ks)
True
```

Testing we have sufficient precision for the golden ratio (trac ticket #12163), note that the decimal point adds 1 to the string length:

```
sage: len(str(n(golden_ratio, digits=5000)))
5001
sage: len(str(n(golden_ratio, digits=5000000)))  # long time (4s on sage.math, 2012)
5000001
```

Check that trac ticket #14778 is fixed:

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

`sage.misc.functional.`**`ngens`**`(x)`

Returns the number of generators of x.

EXAMPLES:

```
sage: R.<x,y> = SR[]; R
Multivariate Polynomial Ring in x, y over Symbolic Ring
sage: ngens(R)
2
sage: A = AbelianGroup(5, [5,5,7,8,9])
sage: ngens(A)
5
sage: ngens(ZZ)
1
```

`sage.misc.functional.`**`norm`**`(x)`

Returns the norm of `x`.

For matrices and vectors, this returns the L2-norm. The L2-norm of a vector $\mathbf{v} = (v_1, v_2, \dots, v_n)$, also called the Euclidean norm, is defined as

$$|\mathbf{v}| = \sqrt{\sum_{i=1}^{n} |v_i|^2}$$

where $|v_i|$ is the complex modulus of v_i. The Euclidean norm is often used for determining the distance between two points in two- or three-dimensional space.

For complex numbers, the function returns the field norm. If $c = a + bi$ is a complex number, then the norm of

c is defined as the product of c and its complex conjugate

$$\text{norm}(c) = \text{norm}(a + bi) = c \cdot \bar{c} = a^2 + b^2.$$

The norm of a complex number is different from its absolute value. The absolute value of a complex number is defined to be the square root of its norm. A typical use of the complex norm is in the integral domain $\mathbf{Z}[i]$ of Gaussian integers, where the norm of each Gaussian integer $c = a + bi$ is defined as its complex norm.

See also:

- `sage.matrix.matrix2.Matrix.norm()`
- `sage.modules.free_module_element.FreeModuleElement.norm()`
- `sage.rings.complex_double.ComplexDoubleElement.norm()`
- `sage.rings.complex_number.ComplexNumber.norm()`
- `sage.symbolic.expression.Expression.norm()`

EXAMPLES:

The norm of vectors:

```
sage: z = 1 + 2*I
sage: norm(vector([z]))
sqrt(5)
sage: v = vector([-1,2,3])
sage: norm(v)
sqrt(14)
sage: _ = var("a b c d")
sage: v = vector([a, b, c, d])
sage: norm(v)
sqrt(abs(a)^2 + abs(b)^2 + abs(c)^2 + abs(d)^2)
```

The norm of matrices:

```
sage: z = 1 + 2*I
sage: norm(matrix([[z]]))
2.23606797749979
sage: M = matrix(ZZ, [[1,2,4,3], [-1,0,3,-10]])
sage: norm(M)  # abs tol 1e-14
10.690331129154467
sage: norm(CDF(z))
5.0
sage: norm(CC(z))
5.00000000000000
```

The norm of complex numbers:

```
sage: z = 2 - 3*I
sage: norm(z)
13
sage: a = randint(-10^10, 100^10)
sage: b = randint(-10^10, 100^10)
sage: z = a + b*I
sage: bool(norm(z) == a^2 + b^2)
True
```

The complex norm of symbolic expressions:

```
sage: a, b, c = var("a, b, c")
sage: assume((a, 'real'), (b, 'real'), (c, 'real'))
sage: z = a + b*I
sage: bool(norm(z).simplify() == a^2 + b^2)
True
sage: norm(a + b).simplify()
a^2 + 2*a*b + b^2
sage: v = vector([a, b, c])
sage: bool(norm(v).simplify() == sqrt(a^2 + b^2 + c^2))
True
sage: forget()
```

`sage.misc.functional.`**`numerator`**(*x*)

Returns the numerator of x.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: F = FractionField(R)
sage: r = (x+1)/(x-1)
sage: numerator(r)
x + 1
sage: numerator(17/11111)
17
```

`sage.misc.functional.`**`numerical_approx`**(*x*, *prec=None*, *digits=None*, *algorithm=None*)

Returns a numerical approximation of an object `x` with at least `prec` bits (or decimal `digits`) of precision.

Note: Both upper case `N` and lower case `n` are aliases for `numerical_approx()`, and all three may be used as methods.

INPUT:

- `x` - an object that has a numerical_approx method, or can be coerced into a real or complex field
- `prec` (optional) - an integer (bits of precision)
- `digits` (optional) - an integer (digits of precision)
- `algorithm` (optional) - a string specifying the algorithm to use for functions that implement more than one

If neither the `prec` or `digits` are specified, the default is 53 bits of precision. If both are specified, then `prec` is used.

EXAMPLES:

```
sage: numerical_approx(pi, 10)
3.1
sage: numerical_approx(pi, digits=10)
3.141592654
sage: numerical_approx(pi^2 + e, digits=20)
12.587886229548403854
sage: n(pi^2 + e)
12.5878862295484
sage: N(pi^2 + e)
12.5878862295484
sage: n(pi^2 + e, digits=50)
12.587886229548403854194778471228813633070946500941
sage: a = CC(-5).n(prec=40)
```

```
sage: b = ComplexField(40)(-5)
sage: a == b
True
sage: parent(a) is parent(b)
True
sage: numerical_approx(9)
9.00000000000000
```

You can also usually use method notation.

```
sage: (pi^2 + e).n()
12.5878862295484
sage: (pi^2 + e).N()
12.5878862295484
sage: (pi^2 + e).numerical_approx()
12.5878862295484
```

Vectors and matrices may also have their entries approximated.

```
sage: v = vector(RDF, [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)

sage: v = vector(CDF, [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: _.parent()
Vector space of dimension 3 over Complex Field with 53 bits of precision
sage: v.n(prec=20)
(1.0000, 2.0000, 3.0000)

sage: u = vector(QQ, [1/2, 1/3, 1/4])
sage: n(u, prec=15)
(0.5000, 0.3333, 0.2500)
sage: n(u, digits=5)
(0.50000, 0.33333, 0.25000)

sage: v = vector(QQ, [1/2, 0, 0, 1/3, 0, 0, 0, 1/4], sparse=True)
sage: u = v.numerical_approx(digits=4)
sage: u.is_sparse()
True
sage: u
(0.5000, 0.0000, 0.0000, 0.3333, 0.0000, 0.0000, 0.0000, 0.2500)

sage: A = matrix(QQ, 2, 3, range(6))
sage: A.n()
[0.000000000000000  1.00000000000000  2.00000000000000]
[ 3.00000000000000  4.00000000000000  5.00000000000000]

sage: B = matrix(Integers(12), 3, 8, srange(24))
sage: N(B, digits=2)
[0.00  1.0  2.0  3.0  4.0  5.0  6.0  7.0]
[ 8.0  9.0  10.  11. 0.00  1.0  2.0  3.0]
[ 4.0  5.0  6.0  7.0  8.0  9.0  10.  11.]
```

Internally, numerical approximations of real numbers are stored in base-2. Therefore, numbers which look the same in their decimal expansion might be different:

```
sage: x=N(pi, digits=3); x
3.14
sage: y=N(3.14, digits=3); y
3.14
sage: x==y
False
sage: x.str(base=2)
'11.001001000100'
sage: y.str(base=2)
'11.001000111101'
```

Increasing the precision of a floating point number is not allowed:

```
sage: CC(-5).n(prec=100)
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 100 bits, use at most 53 bits
sage: n(1.3r, digits=20)
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 70 bits, use at most 53 bits
sage: RealField(24).pi().n()
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 53 bits, use at most 24 bits
```

As an exceptional case, `digits=1` usually leads to 2 digits (one significant) in the decimal output (see trac ticket #11647):

```
sage: N(pi, digits=1)
3.2
sage: N(pi, digits=2)
3.1
sage: N(100*pi, digits=1)
320.
sage: N(100*pi, digits=2)
310.
```

In the following example, `pi` and `3` are both approximated to two bits of precision and then subtracted, which kills two bits of precision:

```
sage: N(pi, prec=2)
3.0
sage: N(3, prec=2)
3.0
sage: N(pi - 3, prec=2)
0.00
```

TESTS:

```
sage: numerical_approx(I)
1.00000000000000*I
sage: x = QQ['x'].gen()
sage: F.<k> = NumberField(x^2+2, embedding=sqrt(CC(2))*CC.0)
sage: numerical_approx(k)
1.41421356237309*I

sage: type(numerical_approx(CC(1/2)))
<type 'sage.rings.complex_number.ComplexNumber'>
```

The following tests trac ticket #10761, in which `n()` would break when called on complex-valued algebraic numbers.

```
sage: E = matrix(3, [3,1,6,5,2,9,7,3,13]).eigenvalues(); E
[18.16815365088822?, -0.08407682544410650? - 0.2190261484802906?*I, -0.08407682544410650? + 0.21
sage: E[1].parent()
Algebraic Field
sage: [a.n() for a in E]
[18.1681536508882, -0.0840768254441065 - 0.219026148480291*I, -0.0840768254441065 + 0.2190261484
```

Make sure we've rounded up log(10,2) enough to guarantee sufficient precision (trac ticket #10164):

```
sage: ks = 4*10**5, 10**6
sage: check_str_length = lambda k: len(str(numerical_approx(1+10**-k,digits=k+1)))-1 >= k+1
sage: check_precision = lambda k: numerical_approx(1+10**-k,digits=k+1)-1 > 0
sage: all(check_str_length(k) and check_precision(k) for k in ks)
True
```

Testing we have sufficient precision for the golden ratio (trac ticket #12163), note that the decimal point adds 1 to the string length:

```
sage: len(str(n(golden_ratio, digits=5000)))
5001
sage: len(str(n(golden_ratio, digits=5000000)))  # long time (4s on sage.math, 2012)
5000001
```

Check that trac ticket #14778 is fixed:

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

`sage.misc.functional.`**`objgen`**(*x*)

EXAMPLES:

```
sage: R, x = objgen(FractionField(QQ['x']))
sage: R
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: x
x
```

`sage.misc.functional.`**`objgens`**(*x*)

EXAMPLES:

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

`sage.misc.functional.`**`one`**(*R*)

Returns the one element of the ring R.

EXAMPLES:

```
sage: one(RR)
doctest:...: DeprecationWarning: one(R) is deprecated, use R.one() or R(1) instead
See http://trac.sagemath.org/17158 for details.
1.00000000000000
sage: R.<x> = PolynomialRing(QQ)
sage: one(R)*x == x
True
sage: one(R) in R
```

```
True
```

sage.misc.functional.**order**(*x*)

Returns the order of x. If x is a ring or module element, this is the additive order of x.

EXAMPLES:

```
sage: C = CyclicPermutationGroup(10)
sage: order(C)
10
sage: F = GF(7)
sage: order(F)
7
```

sage.misc.functional.**quo**(*x, y, *args, **kwds*)

Returns the quotient object x/y, e.g., a quotient of numbers or of a polynomial ring x by the ideal generated by y, etc.

EXAMPLES:

```
sage: quotient(5,6)
5/6
sage: quotient(5.,6.)
0.833333333333333
sage: R.<x> = ZZ[]; R
Univariate Polynomial Ring in x over Integer Ring
sage: I = Ideal(R, x^2+1)
sage: quotient(R, I)
Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 + 1
```

sage.misc.functional.**quotient**(*x, y, *args, **kwds*)

Returns the quotient object x/y, e.g., a quotient of numbers or of a polynomial ring x by the ideal generated by y, etc.

EXAMPLES:

```
sage: quotient(5,6)
5/6
sage: quotient(5.,6.)
0.833333333333333
sage: R.<x> = ZZ[]; R
Univariate Polynomial Ring in x over Integer Ring
sage: I = Ideal(R, x^2+1)
sage: quotient(R, I)
Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 + 1
```

sage.misc.functional.**rank**(*x*)

Returns the rank of x.

EXAMPLES: We compute the rank of a matrix:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: rank(A)
2
```

We compute the rank of an elliptic curve:

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

`sage.misc.functional.`**`regulator`**`(`*x*`)`

Returns the regulator of x.

EXAMPLES:

```
sage: regulator(NumberField(x^2-2, 'a'))
0.881373587019543
sage: regulator(EllipticCurve('11a'))
1.00000000000000
```

`sage.misc.functional.`**`round`**`(`*x*, *ndigits=0*`)`

round(number[, ndigits]) - double-precision real number

Round a number to a given precision in decimal digits (default 0 digits). If no precision is specified this just calls the element's .round() method.

EXAMPLES:

```
sage: round(sqrt(2),2)
1.41
sage: q = round(sqrt(2),5); q
1.41421
sage: type(q)
<type 'sage.rings.real_double.RealDoubleElement'>
sage: q = round(sqrt(2)); q
1
sage: type(q)
<type 'sage.rings.integer.Integer'>
sage: round(pi)
3
sage: b = 5.4999999999999999
sage: round(b)
5
```

Since we use floating-point with a limited range, some roundings can't be performed:

```
sage: round(sqrt(Integer('1'*1000)),2)
+infinity
```

IMPLEMENTATION: If ndigits is specified, it calls Python's builtin round function, and converts the result to a real double field element. Otherwise, it tries the argument's .round() method; if that fails, it reverts to the builtin round function, converted to a real double field element.

Note: This is currently slower than the builtin round function, since it does more work - i.e., allocating an RDF element and initializing it. To access the builtin version do `import __builtin__; __builtin__.round`.

`sage.misc.functional.`**`show`**`(`*x*, **args*, ***kwds*`)`

Show a graphics object x.

For additional ways to show objects in the notebook, look at the methods on the html object. For example, html.table will produce an HTML table from a nested list.

OPTIONAL INPUT:

- `filename` - (default: None) string

SOME OF THESE MAY APPLY:

- `dpi` - dots per inch
- `figsize`- [width, height] (same for square aspect)

- `axes` - (default: True)
- `fontsize` - positive integer
- `frame` - (default: False) draw a MATLAB-like frame around the image

EXAMPLES:

```
sage: show(graphs(3))
sage: show(list(graphs(3)))
```

`sage.misc.functional.`**`squarefree_part`**`(`*x*`)`

Returns the square free part of x, i.e., a divisor z such that $x = zy^2$, for a perfect square y^2.

EXAMPLES:

```
sage: squarefree_part(100)
1
sage: squarefree_part(12)
3
sage: squarefree_part(10)
10
sage: squarefree_part(216r) # see #8976
6

sage: x = QQ['x'].0
sage: S = squarefree_part(-9*x*(x-6)^7*(x-3)^2); S
-9*x^2 + 54*x
sage: S.factor()
(-9) * (x - 6) * x

sage: f = (x^3 + x + 1)^3*(x-1); f
x^10 - x^9 + 3*x^8 + 3*x^5 - 2*x^4 - x^3 - 2*x - 1
sage: g = squarefree_part(f); g
x^4 - x^3 + x^2 - 1
sage: g.factor()
(x - 1) * (x^3 + x + 1)
```

`sage.misc.functional.`**`symbolic_sum`**`(`*expression*, **args*, ***kwds*`)`

Returns the symbolic sum $\sum_{v=a}^{b} expression$ with respect to the variable v with endpoints a and b.

INPUT:

- `expression` - a symbolic expression
- `v` - a variable or variable name
- `a` - lower endpoint of the sum
- `b` - upper endpoint of the sum
- `algorithm` - (default: `'maxima'`) one of
  - `'maxima'` - use Maxima (the default)
  - `'maple'` - (optional) use Maple
  - `'mathematica'` - (optional) use Mathematica
  - `'giac'` - (optional) use Giac

EXAMPLES:

```
sage: k, n = var('k,n')
sage: sum(k, k, 1, n).factor()
1/2*(n + 1)*n
```

```
sage: sum(1/k^4, k, 1, oo)
1/90*pi^4
```

```
sage: sum(1/k^5, k, 1, oo)
zeta(5)
```

> **Warning:** This function only works with symbolic expressions. To sum any other objects like list elements or function return values, please use python summation, see http://docs.python.org/library/functions.html#sum
> In particular, this does not work:
>
> ```
> sage: n = var('n')
> sage: list=[1,2,3,4,5]
> sage: sum(list[n],n,0,3)
> Traceback (most recent call last):
> ...
> TypeError: unable to convert n to an integer
> ```
>
> Use python `sum()` instead:
>
> ```
> sage: sum(list[n] for n in range(4))
> 10
> ```
>
> Also, only a limited number of functions are recognized in symbolic sums:
>
> ```
> sage: sum(valuation(n,2),n,1,5)
> Traceback (most recent call last):
> ...
> TypeError: unable to convert n to an integer
> ```
>
> Again, use python `sum()`:
>
> ```
> sage: sum(valuation(n+1,2) for n in range(5))
> 3
> ```
>
> (now back to the Sage `sum` examples)

A well known binomial identity:

```
sage: sum(binomial(n,k), k, 0, n)
2^n
```

The binomial theorem:

```
sage: x, y = var('x, y')
sage: sum(binomial(n,k) * x^k * y^(n-k), k, 0, n)
(x + y)^n
```

```
sage: sum(k * binomial(n, k), k, 1, n)
2^(n - 1)*n
```

```
sage: sum((-1)^k*binomial(n,k), k, 0, n)
0
```

```
sage: sum(2^(-k)/(k*(k+1)), k, 1, oo)
-log(2) + 1
```

Another binomial identity (trac ticket #7952):

```
sage: t,k,i = var('t,k,i')
sage: sum(binomial(i+t,t),i,0,k)
binomial(k + t + 1, t + 1)
```

Summing a hypergeometric term:

```
sage: sum(binomial(n, k) * factorial(k) / factorial(n+1+k), k, 0, n)
1/2*sqrt(pi)/factorial(n + 1/2)
```

We check a well known identity:

```
sage: bool(sum(k^3, k, 1, n) == sum(k, k, 1, n)^2)
True
```

A geometric sum:

```
sage: a, q = var('a, q')
sage: sum(a*q^k, k, 0, n)
(a*q^(n + 1) - a)/(q - 1)
```

The geometric series:

```
sage: assume(abs(q) < 1)
sage: sum(a*q^k, k, 0, oo)
-a/(q - 1)
```

A divergent geometric series. Don't forget to forget your assumptions:

```
sage: forget()
sage: assume(q > 1)
sage: sum(a*q^k, k, 0, oo)
Traceback (most recent call last):
...
ValueError: Sum is divergent.
```

This summation only Mathematica can perform:

```
sage: sum(1/(1+k^2), k, -oo, oo, algorithm = 'mathematica')     # optional - mathematica
pi*coth(pi)
```

Use Maple as a backend for summation:

```
sage: sum(binomial(n,k)*x^k, k, 0, n, algorithm = 'maple')      # optional - maple
(x + 1)^n
```

Python ints should work as limits of summation (trac ticket #9393):

```
sage: sum(x, x, 1r, 5r)
15
```

Note:

1. Sage can currently only understand a subset of the output of Maxima, Maple and Mathematica, so even if the chosen backend can perform the summation the result might not be convertable into a Sage expression.

sage.misc.functional.**transpose**(*x*)

Returns the transpose of x.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: transpose(A)
[1 4 7]
[2 5 8]
[3 6 9]
```

sage.misc.functional.**xinterval**(*a*, *b*)

Iterator over the integers between a and b, *inclusive*.

EXAMPLES:

```
sage: I = xinterval(2,5); I
xrange(2, 6)
sage: 5 in I
True
sage: 6 in I
False
```

sage.misc.functional.**zero**(*R*)

Returns the zero element of the ring R.

EXAMPLES:

```
sage: zero(RR)
doctest:...: DeprecationWarning: zero(R) is deprecated, use R.zero() or R(0) instead
See http://trac.sagemath.org/17158 for details.
0.000000000000000
sage: R.<x> = PolynomialRing(QQ)
sage: zero(R) in R
True
sage: zero(R)*x == zero(R)
True
```

CHAPTER

TWENTYTHREE

HTML TYPESETTING FOR THE NOTEBOOK

class `sage.misc.html.HTML`

`eval`(*s*, *globals=None*, *locals=None*)

Return an html representation for an object `s`.

If `s` has a method `_html_()`, call that. Otherwise, call `math_parse()` on `str(s)`, evaluate any variables in the result, and add some html preamble and postamble.

In any case, *print* the resulting html string. This method always *returns* an empty string.

EXAMPLES:

```
sage: html.eval('<hr>')
<html><font color='black'><hr></font></html>
''
```

`iframe`(*url*, *height=400*, *width=800*)

Put an existing web page into a worksheet.

INPUT:

- `url` – a url string, either with or without URI scheme (defaults to "http").
- `height` – the number of pixels for the page height. Defaults to 400.
- `width` – the number of pixels for the page width. Defaults to 800.

OUTPUT:

Opens the url in a worksheet. If the url is a regular web page it will appear in the worksheet. This was originally intended to bring GeoGebra worksheets into Sage, but it can be used for many other purposes.

EXAMPLES:

```
sage: html.iframe("sagemath.org")
<html><font color='black'><iframe height="400" width="800"
src="http://sagemath.org"></iframe></font></html>
sage: html.iframe("http://sagemath.org",30,40)
<html><font color='black'><iframe height="30" width="40"
src="http://sagemath.org"></iframe></font></html>
sage: html.iframe("https://sagemath.org",30)
<html><font color='black'><iframe height="30" width="800"
src="https://sagemath.org"></iframe></font></html>
sage: html.iframe("/home/admin/0/data/filename")
<html><font color='black'><iframe height="400" width="800"
src="/home/admin/0/data/filename"></iframe></font></html>
sage: html.iframe('data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAA'
```

```
... 'AUAAAAFCAYAAACNbyblAAAAHElEQVQI12P4//8/w38GIAXDIBKE0DHxgljNBA'
... 'AO9TXL0Y4OHwAAAABJRU5ErkJggg=="')
<html><font color='black'><iframe height="400" width="800"
src="data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAUAAAAFCAYAAACNbyblAAAAHElEQVQI12P4//8/w
```

AUTHOR:

- Bruce Cohen (2011-06-14)

table (*x*, *header=False*)

Print a nested list as a HTML table. Strings of html will be parsed for math inside dollar and double-dollar signs. 2D graphics will be displayed in the cells. Expressions will be latexed.

INPUT:

- `x` – a list of lists (i.e., a list of table rows)
- `header` – a row of headers. If `True`, then the first row of the table is taken to be the header.

EXAMPLES:

```
sage: html.table([(i, j, i == j) for i in [0..1] for j in [0..1]])
<html>
<div class="notruncate">
<table class="table_form">
<tbody>
<tr class ="row-a">
<td><script type="math/tex">0</script></td>
<td><script type="math/tex">0</script></td>
<td><script type="math/tex">\mathrm{True}</script></td>
</tr>
<tr class ="row-b">
<td><script type="math/tex">0</script></td>
<td><script type="math/tex">1</script></td>
<td><script type="math/tex">\mathrm{False}</script></td>
</tr>
<tr class ="row-a">
<td><script type="math/tex">1</script></td>
<td><script type="math/tex">0</script></td>
<td><script type="math/tex">\mathrm{False}</script></td>
</tr>
<tr class ="row-b">
<td><script type="math/tex">1</script></td>
<td><script type="math/tex">1</script></td>
<td><script type="math/tex">\mathrm{True}</script></td>
</tr>
</tbody>
</table>
</div>
</html>

sage: html.table([(x,n(sin(x), digits=2)) for x in [0..3]], header = ["$x$", "$\sin(x)$"])
<html>
<div class="notruncate">
<table class="table_form">
<tbody>
<tr>
<th><script type="math/tex">x</script></th>
<th><script type="math/tex">\sin(x)</script></th>
</tr>
```

```
<tr class ="row-a">
<td><script type="math/tex">0</script></td>
<td><script type="math/tex">0.00</script></td>
</tr>
<tr class ="row-b">
<td><script type="math/tex">1</script></td>
<td><script type="math/tex">0.84</script></td>
</tr>
<tr class ="row-a">
<td><script type="math/tex">2</script></td>
<td><script type="math/tex">0.91</script></td>
</tr>
<tr class ="row-b">
<td><script type="math/tex">3</script></td>
<td><script type="math/tex">0.14</script></td>
</tr>
</tbody>
</table>
</div>
</html>
```

class `sage.misc.html.`**`HTMLExpr`**

Bases: `str`

A class for HTML expression

`sage.misc.html.`**`html`**(*s*, *globals=None*, *locals=None*)

Display the given HTML expression in the notebook.

INPUT:

- `s` – a string

OUTPUT:

- prints a code that embeds HTML in the output.

By default in the notebook an output cell has two parts, first a plain text preformat part, then second a general HTML part (not pre). If you call html(s) at any point then that adds something that will be displayed in the preformated part in html.

EXAMPLES:

```
sage: html('<a href="http://sagemath.org">sagemath</a>')
<html><font color='black'><a href="http://sagemath.org">sagemath</a></font></html>
sage: html('<hr>')
<html><font color='black'><hr></font></html>
```

`sage.misc.html.`**`math_parse`**(*s*)

Turn the HTML-ish string s that can have $$ and $'s in it into pure HTML. See below for a precise definition of what this means.

INPUT:

- `s` – a string

OUTPUT:

- a string.

Do the following:

- Replace all `$ text $`'s by `<script type="math/tex"> text </script>`

- Replace all `$$ text $$`'s by `<script type="math/tex; mode=display"> text </script>`
- Replace all `\ $`'s by `$`'s. Note that in the above two cases nothing is done if the `$` is preceeded by a backslash.
- Replace all `\[ text \]`'s by `<script type="math/tex; mode=display"> text </script>`

EXAMPLES:

```
sage: sage.misc.html.math_parse('This is $2+2$.')
'This is <script type="math/tex">2+2</script>.'
sage: sage.misc.html.math_parse('This is $$2+2$$.')
'This is <script type="math/tex; mode=display">2+2</script>.'
sage: sage.misc.html.math_parse('This is \\[2+2\\].')
'This is <script type="math/tex; mode=display">2+2</script>.'
sage: sage.misc.html.math_parse(r'This is \[2+2\].')
'This is <script type="math/tex; mode=display">2+2</script>.'
```

TESTS:

```
sage: sage.misc.html.math_parse(r'This $$is $2+2$.')
'This $$is <script type="math/tex">2+2</script>.'
```

CHAPTER

TWENTYFOUR

TABLES

Display a rectangular array as a table, either in plain text, LaTeX, or html. See the documentation for `table` for details and examples.

AUTHORS:

- John H. Palmieri (2012-11)

class `sage.misc.table.`**`table`**(*rows=None*, *columns=None*, *header_row=False*, *header_column=False*, *frame=False*, *align='left'*)

Bases: `sage.structure.sage_object.SageObject`

Display a rectangular array as a table, either in plain text, LaTeX, or html.

INPUTS:

- `rows` (default `None`) - a list of lists (or list of tuples, etc.), containing the data to be displayed.
- `columns` (default `None`) - a list of lists (etc.), containing the data to be displayed, but stored as columns. Set either `rows` or `columns`, but not both.
- `header_row` (default `False`) - if `True`, first row is highlighted.
- `header_column` (default `False`) - if `True`, first column is highlighted.
- `frame` (default `False`) - if `True`, put a box around each cell.
- `align` (default 'left') - the alignment of each entry: either 'left', 'center', or 'right'

EXAMPLES:

```
sage: rows = [['a', 'b', 'c'], [100,2,3], [4,5,60]]
sage: table(rows)
  a     b   c
  100   2   3
  4     5   60
sage: latex(table(rows))
\begin{tabular}{lll}
a & b & c \\
$100$ & $2$ & $3$ \\
$4$ & $5$ & $60$ \\
\end{tabular}
```

If `header_row` is `True`, then the first row is highlighted. If `header_column` is `True`, then the first column is highlighted. If `frame` is `True`, then print a box around every "cell".

```
sage: table(rows, header_row=True)
  a     b   c
+-----+---+----+
  100   2   3
  4     5   60
```

```
sage: latex(table(rows, header_row=True))
\begin{tabular}{lll}
a & b & c \\ \hline
$100$ & $2$ & $3$ \\
$4$ & $5$ & $60$ \\
\end{tabular}
sage: table(rows=rows, frame=True)
+-----+---+----+
| a   | b | c  |
+-----+---+----+
| 100 | 2 | 3  |
+-----+---+----+
| 4   | 5 | 60 |
+-----+---+----+
sage: latex(table(rows=rows, frame=True))
\begin{tabular}{|l|l|l|} \hline
a & b & c \\ \hline
$100$ & $2$ & $3$ \\ \hline
$4$ & $5$ & $60$ \\ \hline
\end{tabular}
sage: table(rows, header_column=True, frame=True)
+-----++---+----+
| a   || b | c  |
+-----++---+----+
| 100 || 2 | 3  |
+-----++---+----+
| 4   || 5 | 60 |
+-----++---+----+
sage: latex(table(rows, header_row=True, frame=True))
\begin{tabular}{|l|l|l|} \hline
a & b & c \\ \hline \hline
$100$ & $2$ & $3$ \\ \hline
$4$ & $5$ & $60$ \\ \hline
\end{tabular}
sage: table(rows, header_column=True)
  a   | b   c
  100 | 2   3
  4   | 5   60
```

The argument `header_row` can, instead of being `True` or `False`, be the contents of the header row, so that `rows` consists of the data, while `header_row` is the header information. The same goes for `header_column`. Passing lists for both arguments simultaneously is not supported.

```
sage: table([(x,n(sin(x), digits=2)) for x in [0..3]], header_row=["$x$", "$\sin(x)$"], frame=Tr
+-----+-----------+
| $x$ | $\sin(x)$ |
+=====+===========+
| 0   | 0.00      |
+-----+-----------+
| 1   | 0.84      |
+-----+-----------+
| 2   | 0.91      |
+-----+-----------+
| 3   | 0.14      |
+-----+-----------+
```

You can create the transpose of this table in several ways, for example, "by hand," that is, changing the data defining the table:

```
sage: table(rows=[[x for x in [0..3]], [n(sin(x), digits=2) for x in [0..3]]], header_column=['$
+-----------++------+------+------+------+
| $x$       || 0    | 1    | 2    | 3    |
+-----------++------+------+------+------+
| $\sin(x)$ || 0.00 | 0.84 | 0.91 | 0.14 |
+-----------++------+------+------+------+
```

or by passing the original data as the `columns` of the table and using `header_column` instead of `header_row`:

```
sage: table(columns=[(x,n(sin(x), digits=2)) for x in [0..3]], header_column=['$x$', '$\sin(x)$'
+-----------++------+------+------+------+
| $x$       || 0    | 1    | 2    | 3    |
+-----------++------+------+------+------+
| $\sin(x)$ || 0.00 | 0.84 | 0.91 | 0.14 |
+-----------++------+------+------+------+
```

or by taking the `transpose()` of the original table:

```
sage: table(rows=[(x,n(sin(x), digits=2)) for x in [0..3]], header_row=['$x$', '$\sin(x)$'], fra
+-----------++------+------+------+------+
| $x$       || 0    | 1    | 2    | 3    |
+-----------++------+------+------+------+
| $\sin(x)$ || 0.00 | 0.84 | 0.91 | 0.14 |
+-----------++------+------+------+------+
```

In either plain text or LaTeX, entries in tables can be aligned to the left (default), center, or right:

```
sage: table(rows, align='left')
  a     b   c
  100   2   3
  4     5   60
sage: table(rows, align='center')
  a    b   c
 100   2   3
  4    5   60
sage: table(rows, align='right', frame=True)
+-----+---+----+
|   a | b |  c |
+-----+---+----+
| 100 | 2 |  3 |
+-----+---+----+
|   4 | 5 | 60 |
+-----+---+----+
```

To print HTML, use either `table(...)._html_()` or `html(table(...))`:

```
sage: html(table([["$x$", "$\sin(x)$"]] + [(x,n(sin(x), digits=2)) for x in [0..3]], header_row=
<html>
<div class="notruncate">
<table border="1" class="table_form">
<tbody>
<tr>
<th><script type="math/tex">x</script></th>
<th><script type="math/tex">\sin(x)</script></th>
</tr>
<tr class ="row-a">
<td><script type="math/tex">0</script></td>
<td><script type="math/tex">0.00</script></td>
</tr>
```

```
<tr class ="row-b">
<td><script type="math/tex">1</script></td>
<td><script type="math/tex">0.84</script></td>
</tr>
<tr class ="row-a">
<td><script type="math/tex">2</script></td>
<td><script type="math/tex">0.91</script></td>
</tr>
<tr class ="row-b">
<td><script type="math/tex">3</script></td>
<td><script type="math/tex">0.14</script></td>
</tr>
</tbody>
</table>
</div>
</html>
```

It is an error to specify both `rows` and `columns`:

```
sage: table(rows=[[1,2,3], [4,5,6]], columns=[[0,0,0], [0,0,1024]])
Traceback (most recent call last):
...
ValueError: Don't set both 'rows' and 'columns' when defining a table.

sage: table(columns=[[0,0,0], [0,0,1024]])
0 0
0 0
0 1024
```

Note that if `rows` is just a list or tuple, not nested, then it is treated as a single row:

```
sage: table([1,2,3])
1   2   3
```

Also, if you pass a non-rectangular array, the longer rows or columns get truncated:

```
sage: table([[1,2,3,7,12], [4,5]])
1   2
4   5
sage: table(columns=[[1,2,3], [4,5,6,7]])
1   4
2   5
3   6
```

TESTS:

```
sage: TestSuite(table([["$x$", "$\sin(x)$"]] + [(x,n(sin(x), digits=2)) for x in [0..3]], header
```

options(***kwds*)

With no arguments, return the dictionary of options for this table. With arguments, modify options.

INPUTS:

- `header_row` - if True, first row is highlighted.
- `header_column` - if True, first column is highlighted.
- `frame` - if True, put a box around each cell.
- `align` - the alignment of each entry: either 'left', 'center', or 'right'

EXAMPLES:

```
sage: T = table([['a', 'b', 'c'], [1,2,3]])
sage: T.options()['align'], T.options()['frame']
('left', False)
sage: T.options(align='right', frame=True)
sage: T.options()['align'], T.options()['frame']
('right', True)
```

Note that when first initializing a table, `header_row` or `header_column` can be a list. In this case, during the initialization process, the header is merged with the rest of the data, so changing the header option later using `table.options(...)` doesn't affect the contents of the table, just whether the row or column is highlighed. When using this `options()` method, no merging of data occurs, so here `header_row` and `header_column` should just be `True` or `False`, not a list.

```
sage: T = table([[1,2,3], [4,5,6]], header_row=['a', 'b', 'c'], frame=True)
sage: T
+---+---+---+
| a | b | c |
+===+===+===+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
sage: T.options(header_row=False)
sage: T
+---+---+---+
| a | b | c |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
```

If you do specify a list for `header_row`, an error is raised:

```
sage: T.options(header_row=['x', 'y', 'z'])
Traceback (most recent call last):
...
TypeError: header_row should be either True or False.
```

transpose()

Return a table which is the transpose of this one: rows and columns have been interchanged. Several of the properties of the original table are preserved: whether a frame is present and any alignment setting. On the other hand, header rows are converted to header columns, and vice versa.

EXAMPLES:

```
sage: T = table([[1,2,3], [4,5,6]])
sage: T.transpose()
  1   4
  2   5
  3   6
sage: T = table([[1,2,3], [4,5,6]], header_row=['x', 'y', 'z'], frame=True)
sage: T.transpose()
+---++---+---+
| x || 1 | 4 |
+---++---+---+
| y || 2 | 5 |
+---++---+---+
| z || 3 | 6 |
```

```
+---++---+---+
```

CHAPTER

TWENTYFIVE

LOGGING OF SAGE SESSIONS

TODO: Pressing "control-D" can mess up the I/O sequence because of a known bug.

You can create a log of your Sage session as a web page and/or as a latex document. Just type `log_html()` to create an HTML log, or `log_dvi()` to create a dvi (LaTeX) log. Your complete session so far up until when you type the above command will be logged, along with any future input. Thus you can view the log system as a way to print or view your entire session so far, along with a way to see nicely typeset incremental updates as you work.

If `L=log_dvi()` or `L=log_html()` is a logger, you can type `L.stop()` and `L.start()` to stop and start logging.

The environment variables `BROWSER` and `DVI_VIEWER` determine which web browser or dvi viewer is used to display your running log.

For both log systems you must have a TeX system installed on your computer. For HTML logging, you must have the convert command, which comes with the free ImageMagick tools.

Note: The HTML output is done via LaTeX and PNG images right now, sort of like how latex2html works. Obviously it would be interesting to do something using MathML in the long run.

AUTHORS:

- William Stein (2006-02): initial version
- William Stein (2006-02-27): changed html generation so log directory is relocatable (no hardcoded paths).
- William Stein (2006-03-04): changed environment variable to BROWSER.
- Didier Deshommes (2006-05-06): added MathML support; refactored code.
- Dan Drake (2008-03-27): fix bit rotting so that optional directories work, dvi logging works, viewer() command works, remove no-longer-working MathML logger; fix off-by-one problems with IPython history; add text logger; improve documentation about viewers.

class `sage.misc.log.`**`Log`**(*dir=None*, *debug=False*, *viewer=None*)

This is the base logger class. The two classes that you actually instantiate are derived from this one.

`dir`()

Return the directory that contains the log files.

`start`()

Start the logger. To stop use the stop function.

`stop`()

Stop the logger. To restart use the start function.

class `sage.misc.log.`**`log_dvi`**(*dir=None*, *debug=False*, *viewer=None*)

Bases: `sage.misc.log.Log`

Create a running log of your Sage session as a nicely typeset dvi file.

Easy usage: `log_dvi()`

TODO: Pressing "control-D" can mess up the I/O sequence because of a known bug.

Use `L=log_dvi([optional directory])` to create a dvi log. Your complete session so far up until when you type the above command will be logged, along with any future input. Thus you can view the log system as a way to print or view your entire session so far, along with a way to see nicely typeset incremental updates as you work.

If L is a logger, you can type `L.stop()` and `L.start()` to stop and start logging.

The environment variable `DVI_VIEWER` determines which web browser or dvi viewer is used to display your running log. You can also specify a viewer when you start the logger with something like `log_dvi([opt. dir], viewer='xdvi')`.

You must have a LaTeX system installed on your computer and a dvi viewer.

view()

x.__init__(...) initializes x; see help(type(x)) for signature

class `sage.misc.log.`**`log_html`**(*dir=None*, *debug=False*, *viewer=None*)

Bases: `sage.misc.log.Log`

Create a running log of your Sage session as a web page.

Easy usage: `log_html()`

TODO: Pressing "control-D" can mess up the I/O sequence because of a known bug.

Use `L=log_html([optional directory])` to create an HTML log. Your complete session so far up until when you type the above command will be logged, along with any future input. Thus you can view the log system as a way to print or view your entire session so far, along with a way to see nicely typeset incremental updates as you work.

If L is a logger, you can type `L.stop()` and `L.start()` to stop and start logging.

The environment variable `WEB_BROWSER` determines which web browser or dvi viewer is used to display your running log. You can also specify a viewer when you start the logger with something like `log_html([opt. dir], viewer='firefox')`.

You must have a TeX system installed on your computer, and you must have the convert command, which comes with the free ImageMagick tools.

view()

x.__init__(...) initializes x; see help(type(x)) for signature

class `sage.misc.log.`**`log_text`**(*dir=None*, *debug=False*, *viewer=None*)

Bases: `sage.misc.log.Log`

Create a running log of your Sage session as a plain text file.

Easy usage: `log_text()`

TODO: Pressing "control-D" can mess up the I/O sequence because of a known bug.

Use `L=log_text([optional directory])` to create a text log. Your complete session so far up until when you type the above command will be logged, along with any future input. Thus you can view the log system as a way to print or view your entire session so far, along with a way to see incremental updates as you work.

Unlike the html and dvi loggers, this one does not automatically start a viewer unless you specify one; you can do that when you start the logger with something like `log_text([opt. dir], viewer='xterm -e tail -f')`.

If L is a logger, you can type `L.stop()` and `L.start()` to stop and start logging.

view()

x.__init__(...) initializes x; see help(type(x)) for signature

sage.misc.log.**update**()

x.__init__(...) initializes x; see help(type(x)) for signature

CHAPTER

TWENTYSIX

OBJECT PERSISTENCE

You can load and save most Sage object to disk using the load and save member functions and commands.

Note: It is impossible to save certain Sage objects to disk. For example, if x is a MAGMA object, i.e., a wrapper around an object that is defined in MAGMA, there is no way to save x it to disk, since MAGMA doesn't support saving of individual objects to disk.

- Versions: Loading and saving of objects is guaranteed to work even if the version of Python changes. Saved objects can be loaded in future versions of Python. However, if the data structure that defines the object, e.g., in Sage code, changes drastically (or changes name or disappears), then the object might not load correctly or work correctly.
- Objects are zlib compressed for space efficiency.

`sage.misc.persist.`**`db`**`(`*name*`)`

Load object with given name from the Sage database. Use x.db(name) or db_save(x, name) to save objects to the database.

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

`sage.misc.persist.`**`db_save`**`(`*x*, *name=None*`)`

Save x to the Sage database.

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

`sage.misc.persist.`**`load_sage_element`**`(`*cls*, *parent*, *dic_pic*`)`

x.__init__(...) initializes x; see help(type(x)) for signature

`sage.misc.persist.`**`load_sage_object`**`(`*cls*, *dic*`)`

x.__init__(...) initializes x; see help(type(x)) for signature

CHAPTER

TWENTYSEVEN

THE UNKNOWN TRUTH VALUE

AUTHORS:

- Florent Hivert (2010): initial version.

class `sage.misc.unknown.`**`UnknownClass`**

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.sage_object.SageObject`

TESTS:

```
sage: TestSuite(Unknown).run()
```

CHAPTER

TWENTYEIGHT

SUPPORT FOR PERSISTENT FUNCTIONS IN .SAGE FILES

Persistent functions are functions whose values are stored on disk so they do not have to be recomputed.

The inputs to the function must be hashable (so lists are not allowed). Though a hash is used, in the incredibly unlikely event that a hash collision occurs, your function will not return an incorrect result because of this (though the cache might not be used either).

This is meant to be used from `.sage` files, not from library `.py` files.

To use this disk caching mechanism, just put `@func_persist` right before your function definition. For example,

```
@func_persist
def bern(n):
    "Return the n-th Bernoulli number, caching the result to disk."
    return bernoulli(n)
```

You can then use the function `bern` as usual, except it will almost instantly return values that have already been computed, even if you quit and restart.

The disk cache files are stored by default in the subdirectory `func_persist` of the current working directory, with one file for each evaluation of the function.

class `sage.misc.func_persist.`**`func_persist`**(*f*, *dir='func_persist'*)

Put `@func_persist` right before your function definition to cache values it computes to disk.

CHAPTER

TWENTYNINE

EVALUATING A STRING IN SAGE

sage.misc.sage_eval.**sage_eval**(*source*, *locals=None*, *cmds=''*, *preparse=True*)

Obtain a Sage object from the input string by evaluating it using Sage. This means calling eval after preparsing and with globals equal to everything included in the scope of `from sage.all import *`.).

INPUT:

- `source` - a string or object with a _sage_ method
- `locals` - evaluate in namespace of sage.all plus the locals dictionary
- `cmds` - string; sequence of commands to be run before source is evaluated.
- `preparse` - (default: True) if True, preparse the string expression.

EXAMPLES: This example illustrates that preparsing is applied.

```
sage: eval('2^3')
1
sage: sage_eval('2^3')
8
```

However, preparsing can be turned off.

```
sage: sage_eval('2^3', preparse=False)
1
```

Note that you can explicitly define variables and pass them as the second option:

```
sage: x = PolynomialRing(RationalField(),"x").gen()
sage: sage_eval('x^2+1', locals={'x':x})
x^2 + 1
```

This example illustrates that evaluation occurs in the context of `from sage.all import *`. Even though bernoulli has been redefined in the local scope, when calling `sage_eval` the default value meaning of bernoulli is used. Likewise for QQ below.

```
sage: bernoulli = lambda x : x^2
sage: bernoulli(6)
36
sage: eval('bernoulli(6)')
36
sage: sage_eval('bernoulli(6)')
1/42

sage: QQ = lambda x : x^2
sage: QQ(2)
4
sage: sage_eval('QQ(2)')
```

```
2
sage: parent(sage_eval('QQ(2)'))
Rational Field
```

This example illustrates setting a variable for use in evaluation.

```
sage: x = 5
sage: eval('4/3 + x', {'x':25})
26
sage: sage_eval('4/3 + x',  locals={'x':25})
79/3
```

You can also specify a sequence of commands to be run before the expression is evaluated:

```
sage: sage_eval('p', cmds='K.<x> = QQ[]\np = x^2 + 1')
x^2 + 1
```

If you give commands to execute and a dictionary of variables, then the dictionary will be modified by assignments in the commands:

```
sage: vars = {}
sage: sage_eval('None', cmds='y = 3', locals=vars)
sage: vars['y'], parent(vars['y'])
(3, Integer Ring)
```

You can also specify the object to evaluate as a tuple. A 2-tuple is assumed to be a pair of a command sequence and an expression; a 3-tuple is assumed to be a triple of a command sequence, an expression, and a dictionary holding local variables. (In this case, the given dictionary will not be modified by assignments in the commands.)

```
sage: sage_eval(('f(x) = x^2', 'f(3)'))
9
sage: vars = {'rt2': sqrt(2.0)}
sage: sage_eval(('rt2 += 1', 'rt2', vars))
2.41421356237309
sage: vars['rt2']
1.41421356237310
```

This example illustrates how `sage_eval` can be useful when evaluating the output of other computer algebra systems.

```
sage: R.<x> = PolynomialRing(RationalField())
sage: gap.eval('R:=PolynomialRing(Rationals,["x"]);')
'Rationals[x]'
sage: ff = gap.eval('x:=IndeterminatesOfPolynomialRing(R);; f:=x^2+1;'); ff
'x^2+1'
sage: sage_eval(ff, locals={'x':x})
x^2 + 1
sage: eval(ff)
Traceback (most recent call last):
...
RuntimeError: Use ** for exponentiation, not '^', which means xor
in Python, and has the wrong precedence.
```

Here you can see eval simply will not work but `sage_eval` will.

TESTS:

We get a nice minimal error message for syntax errors, that still points to the location of the error (in the input string):

```
sage: sage_eval('RR(22/7]')
Traceback (most recent call last):
...
 File "<string>", line 1
    RR(Integer(22)/Integer(7)]
                             ^
SyntaxError: unexpected EOF while parsing

sage: sage_eval('None', cmds='$x = $y[3] # Does Perl syntax work?')
Traceback (most recent call last):
...
 File "<string>", line 1
    $x = $y[Integer(3)] # Does Perl syntax work?
    ^
SyntaxError: invalid syntax
```

`sage.misc.sage_eval.`**`sageobj`**`(`*x*, *vars=None*`)`

Return a native Sage object associated to x, if possible and implemented.

If the object has an _sage_ method it is called and the value is returned. Otherwise str is called on the object, and all preparsing is applied and the resulting expression is evaluated in the context of `from sage.all import *`. To evaluate the expression with certain variables set, use the vars argument, which should be a dictionary.

EXAMPLES:

```
sage: type(sageobj(gp('34/56')))
<type 'sage.rings.rational.Rational'>
sage: n = 5/2
sage: sageobj(n) is n
True
sage: k = sageobj('Z(8^3/1)', {'Z':ZZ}); k
512
sage: type(k)
<type 'sage.rings.integer.Integer'>
```

This illustrates interfaces:

```
sage: f = gp('2/3')
sage: type(f)
<class 'sage.interfaces.gp.GpElement'>
sage: f._sage_()
2/3
sage: type(f._sage_())
<type 'sage.rings.rational.Rational'>
sage: a = gap(939393/2433)
sage: a._sage_()
313131/811
sage: type(a._sage_())
<type 'sage.rings.rational.Rational'>
```

CHAPTER

THIRTY

SAGE INPUT FORMATTING

This module provides the function `sage_input()` that takes an arbitrary sage value and produces a sequence of commands that, if typed at the `sage:` prompt, will recreate the value. If this is not implemented for a particular value, then an exception is raised instead. This might be useful in understanding a part of Sage, or for debugging. For instance, if you have a value produced in a complicated way in the middle of a debugging session, you could use `sage_input()` to find a simple way to produce the same value. We attempt to produce commands that are readable and idiomatic.:

```
sage: sage_input(3)
3
sage: sage_input((polygen(RR) + RR(pi))^2, verify=True)
# Verified
R.<x> = RR[]
x^2 + 6.2831853071795862*x + 9.869604401089358
```

With `verify=True`, `sage_input()` also verifies the results, by calling `sage_eval()` on the result and verifying that it is equal to the input.:

```
sage: sage_input(GF(2)(1), verify=True)
# Verified
GF(2)(1)
```

We can generate code that works without the preparser, with `preparse=False`; or we can generate code that will work whether or not the preparser is enabled, with `preparse=None`. Generating code with `preparse=False` may be useful to see how to create a certain value in a Python or Cython source file.:

```
sage: sage_input(5, verify=True)
# Verified
5
sage: sage_input(5, preparse=False)
ZZ(5)
sage: sage_input(5, preparse=None)
ZZ(5)
sage: sage_input(5r, verify=True)
# Verified
5r
sage: sage_input(5r, preparse=False)
5
sage: sage_input(5r, preparse=None)
int(5)
```

Adding `sage_input()` support to your own classes is straightforward. You need to add a `_sage_input_()` method which returns a `SageInputExpression` (henceforth abbreviated as SIE) which will reconstruct this instance of your class.

A `_sage_input_` method takes two parameters, conventionally named `sib` and `coerced`. The first argument is a `SageInputBuilder`; it has methods to build SIEs. The second argument, `coerced`, is a boolean. This is only useful if your class is a subclass of `Element` (although it is always present). If `coerced` is `False`, then your method must generate an expression which will evaluate to a value of the correct type with the correct parent. If `coerced` is `True`, then your method may generate an expression of a type that has a canonical coercion to your type; and if `coerced` is 2, then your method may generate an expression of a type that has a conversion to your type.

Let's work through some examples. We'll build a sequence of functions that would be acceptable as `_sage_input_` methods for the `Rational` class.

Here's the first and simplest version.:

```
sage: def qq_sage_input_v1(self, sib, coerced):
....:     return sib(self.numerator())/sib(self.denominator())
```

We see that given a `SageInputBuilder` `sib`, you can construct a SIE for a value `v` simply with `sib(v)`, and you can construct a SIE for a quotient with the division operator. Of course, the other operators also work, and so do function calls, method calls, subscripts, etc.

We'll test with the following code, which you don't need to understand. (It produces a list of 8 results, showing the formatted versions of -5/7 and 3, with the preparser either enabled or disabled and either with or without an automatic coercion to QQ.):

```
sage: from sage.misc.sage_input import SageInputBuilder
sage: def test_qq_formatter(fmt):
....:     results = []
....:     for v in [-5/7, QQ(3)]:
....:         for pp in [False, True]:
....:             for coerced in [False, True]:
....:                 sib = SageInputBuilder(preparse=pp)
....:                 results.append(sib.result(fmt(v, sib, coerced)))
....:     return results

sage: test_qq_formatter(qq_sage_input_v1)
[-ZZ(5)/ZZ(7), -ZZ(5)/ZZ(7), -5/7, -5/7, ZZ(3)/ZZ(1), ZZ(3)/ZZ(1), 3/1, 3/1]
```

Let's try for some shorter, perhaps nicer-looking output. We'll start by getting rid of the `ZZ` in the denominators; even without the preparser, `-ZZ(5)/7 == -ZZ(5)/ZZ(7)`.:

```
sage: def qq_sage_input_v2(self, sib, coerced):
....:     return sib(self.numerator())/sib.int(self.denominator())
```

The `int` method on `SageInputBuilder` returns a SIE for an integer that is always represented in the simple way, without coercions. (So, depending on the preparser mode, it might read in as an `Integer`, an `int`, or a `long`.):

```
sage: test_qq_formatter(qq_sage_input_v2)
[-ZZ(5)/7, -ZZ(5)/7, -5/7, -5/7, ZZ(3)/1, ZZ(3)/1, 3/1, 3/1]
```

Next let's get rid of the divisions by 1. These are more complicated, since if we're not careful we'll get results in ZZ instead of QQ.:

```
sage: def qq_sage_input_v3(self, sib, coerced):
....:     if self.denominator() == 1:
....:         if coerced:
....:             return sib.int(self.numerator())
....:         else:
....:             return sib.name('QQ')(sib.int(self.numerator()))
....:     return sib(self.numerator())/sib.int(self.denominator())
```

We see that the method{name} method gives an SIE representing a sage constant or function.:

```
sage: test_qq_formatter(qq_sage_input_v3)
[-ZZ(5)/7, -ZZ(5)/7, -5/7, -5/7, QQ(3), 3, QQ(3), 3]
```

This is the prettiest output we're going to get, but let's make one further refinement. Other `_sage_input_` methods, like the one for polynomials, analyze the structure of SIEs; they work better (give prettier output) if negations are at the outside. If the above code were used for rationals, then `sage_input(polygen(QQ) - 2/3)` would produce `x + (-2/3)`; if we change to the following code, then we would get `x - 2/3` instead.:

```
sage: def qq_sage_input_v4(self, sib, coerced):
....:     num = self.numerator()
....:     neg = (num < 0)
....:     if neg: num = -num
....:     if self.denominator() == 1:
....:         if coerced:
....:             v = sib.int(num)
....:         else:
....:             v = sib.name('QQ')(sib.int(num))
....:     else:
....:         v = sib(num)/sib.int(self.denominator())
....:     if neg: v = -v
....:     return v

sage: test_qq_formatter(qq_sage_input_v4)
[-ZZ(5)/7, -ZZ(5)/7, -5/7, -5/7, QQ(3), 3, QQ(3), 3]
```

AUTHORS:

- Carl Witty (2008-04): new file
- Vincent Delecroix (2015-02): documentation formatting

class `sage.misc.sage_input.`**`SIE_assign`**(*sib*, *lhs*, *rhs*)

Bases: `sage.misc.sage_input.SageInputExpression`

This class represents an assignment command.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.assign(sib.name('foo').x, sib.name('pi'))
{assign: {getattr: {atomic:foo}.x} {atomic:pi}}
```

class `sage.misc.sage_input.`**`SIE_binary`**(*sib*, *op*, *lhs*, *rhs*)

Bases: `sage.misc.sage_input.SageInputExpression`

This class represents an arithmetic expression with a binary operator and its two arguments, in a `sage_input()` expression tree.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib(3)+5
{binop:+ {atomic:3} {atomic:5}}
```

class sage.misc.sage_input.**SIE_call**(*sib*, *func*, *args*, *kwargs*)
Bases: sage.misc.sage_input.SageInputExpression

This class represents a function-call node in a sage_input() expression tree.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sie = sib.name('GF')
sage: sie(49)
{call: {atomic:GF}({atomic:49})}
```

class sage.misc.sage_input.**SIE_dict**(*sib*, *entries*)
Bases: sage.misc.sage_input.SageInputExpression

This class represents a dict node in a sage_input() expression tree.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.dict([('TeX', RR(pi)), ('Metafont', RR(e))])
{dict: {{atomic:'TeX'}:{call: {atomic:RR}({atomic:3.1415926535897931})}, {atomic:'Metafont'}:{ca
sage: sib.dict({-40:-40, 0:32, 100:212})
{dict: {{unop:- {atomic:40}}:{unop:- {atomic:40}}, {atomic:0}:{atomic:32}, {atomic:100}:{atomic:
```

class sage.misc.sage_input.**SIE_gen**(*sib*, *parent*, *name*)
Bases: sage.misc.sage_input.SageInputExpression

This class represents a named generator of a parent with named generators.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.gen(ZZ['x'])
{gen:x {constr_parent: {subscr: {atomic:ZZ}[{atomic:'x'}]} with gens: ('x',)}}
```

class sage.misc.sage_input.**SIE_gens_constructor**(*sib*, *constr*, *gen_names*, *gens_syntax=None*)
Bases: sage.misc.sage_input.SageInputExpression

This class represents an expression that can create a sage parent with named generators, optionally using the sage preparser generators syntax (like `K.<x> = QQ[]`).

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: qq = sib.name('QQ')
sage: sib.parent_with_gens("some parent", qq['x'],
...                        ('x',), 'QQx',
...                        gens_syntax=sib.empty_subscript(qq))
{constr_parent: {subscr: {atomic:QQ}[{atomic:'x'}]} with gens: ('x',)}
```

class sage.misc.sage_input.**SIE_getattr**(*sib*, *obj*, *attr*)
Bases: sage.misc.sage_input.SageInputExpression

This class represents a getattr node in a `sage_input()` expression tree.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sie = sib.name('CC').gen()
sage: sie
{call: {getattr: {atomic:CC}.gen}()}
```

class `sage.misc.sage_input.`**`SIE_import_name`**(*sib*, *module*, *name*, *alt_name=None*)

Bases: `sage.misc.sage_input.SageInputExpression`

This class represents a name which has been imported from a module.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.import_name('sage.rings.integer', 'make_integer')
{import:sage.rings.integer/make_integer}
sage: sib.import_name('sage.foo', 'happy', 'sad')
{import:sage.foo/happy as sad}
```

class `sage.misc.sage_input.`**`SIE_literal`**(*sib*)

Bases: `sage.misc.sage_input.SageInputExpression`

An abstract base class for `literals` (basically, values which consist of a single token).

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder, SIE_literal

sage: sib = SageInputBuilder()
sage: sie = sib(3)
sage: sie
{atomic:3}
sage: isinstance(sie, SIE_literal)
True
```

class `sage.misc.sage_input.`**`SIE_literal_stringrep`**(*sib*, *n*)

Bases: `sage.misc.sage_input.SIE_literal`

Values in this class are leaves in a `sage_input()` expression tree. Typically they represent a single token, and consist of the string representation of that token. They are used for integer, floating-point, and string literals, and for name expressions.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder, SIE_literal_stringrep

sage: sib = SageInputBuilder()
sage: isinstance(sib(3), SIE_literal_stringrep)
True
sage: isinstance(sib(3.14159, True), SIE_literal_stringrep)
True
sage: isinstance(sib.name('pi'), SIE_literal_stringrep)
True
sage: isinstance(sib(False), SIE_literal_stringrep)
True
```

```
sage: sib(False)
{atomic:False}
```

class sage.misc.sage_input.**SIE_subscript**(*sib*, *coll*, *key*)
Bases: sage.misc.sage_input.SageInputExpression

This class represents a subscript node in a sage_input() expression tree.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sie = sib.name('QQ')['x,y']
sage: sie
{subscr: {atomic:QQ}[{atomic:'x,y'}]}
```

class sage.misc.sage_input.**SIE_tuple**(*sib*, *values*, *is_list*)
Bases: sage.misc.sage_input.SageInputExpression

This class represents a tuple or list node in a sage_input() expression tree.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib((1, 'howdy'))
{tuple: ({atomic:1}, {atomic:'howdy'})}
sage: sib(["lists"])
{list: ({atomic:'lists'})}
```

class sage.misc.sage_input.**SIE_unary**(*sib*, *op*, *operand*)
Bases: sage.misc.sage_input.SageInputExpression

This class represents an arithmetic expression with a unary operator and its argument, in a sage_input() expression tree.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: -sib(256)
{unop:- {atomic:256}}
```

class sage.misc.sage_input.**SageInputAnswer**
Bases: tuple

This class inherits from tuple, so it acts like a tuple when passed to sage_eval(); but it prints as a sequence of commands.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputAnswer
sage: v = SageInputAnswer('x = 22\n', 'x/7'); v
x = 22
x/7
sage: isinstance(v, tuple)
True
sage: v[0]
'x = 22\n'
```

```
sage: v[1]
'x/7'
sage: len(v)
2
sage: v = SageInputAnswer('', 'sin(3.14)', {'sin': math.sin}); v
LOCALS:
  sin: <built-in function sin>
sin(3.14)
sage: v[0]
''
sage: v[1]
'sin(3.14)'
sage: v[2]
{'sin': <built-in function sin>}
```

class sage.misc.sage_input.**SageInputBuilder**(*allow_locals=False*, *preparse=True*)

An instance of this class is passed to `_sage_input_` methods. It keeps track of the current state of the `_sage_input_` process, and contains many utility methods for building `SageInputExpression` objects.

In normal use, instances of `SageInputBuilder` are created internally by `sage_input()`, but it may be useful to create an instance directly for testing or doctesting.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder
```

We can create a `SageInputBuilder`, use it to create some `SageInputExpression` s, and get a result. (As mentioned above, this is only useful for testing or doctesting; normally you would just use `sage_input()`.):

```
sage: sib = SageInputBuilder()
sage: sib.result((sib(3) + sib(4)) * (sib(5) + sib(6)))
(3 + 4)*(5 + 6)
```

assign(*e*, *val*)

Constructs a command that performs the assignment `e=val`.

Can only be used as an argument to the `command` method.

INPUT:

- `e`, `val` – SageInputExpression

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: circular = sib([None])
sage: sib.command(circular, sib.assign(circular[0], circular))
sage: sib.result(circular)
si = [None]
si[0] = si
si
```

cache(*x*, *sie*, *name*)

INPUT:

- `x` - an arbitrary value
- `sie` - a `SageInputExpression`

- name - a requested variable name

Enters x and sie in a cache, so that subsequent calls self(x) will directly return sie. Also, marks the requested name of this sie to be name.

This should almost always be called as part of the method{_sage_input_} method of a parent. It may also be called on values of an arbitrary type, which may be useful if the values are both large and likely to be used multiple times in a single expression.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sie42 = sib(GF(101)(42))
sage: sib.cache(GF(101)(42), sie42, 'the_ultimate_answer')
sage: sib.result(sib(GF(101)(42)) + sib(GF(101)(42)))
the_ultimate_answer = GF(101)(42)
the_ultimate_answer + the_ultimate_answer
```

Note that we don't assign the result to a variable if the value is only used once.:

```
sage: sib = SageInputBuilder()
sage: sie42 = sib(GF(101)(42))
sage: sib.cache(GF(101)(42), sie42, 'the_ultimate_answer')
sage: sib.result(sib(GF(101)(42)) + sib(GF(101)(43)))
GF_101 = GF(101)
GF_101(42) + GF_101(43)
```

command (*v*, *cmd*)

INPUT:

- v, cmd – SageInputExpression

Attaches a command to v, which will be executed before v is used. Multiple commands will be executed in the order added.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: incr_list = sib([])
sage: sib.command(incr_list, incr_list.append(1))
sage: sib.command(incr_list, incr_list.extend([2, 3]))
sage: sib.result(incr_list)
si = []
si.append(1)
si.extend([2, 3])
si
```

dict (*entries*)

Given a dictionary, or a list of (key, value) pairs, produces a SageInputExpression representing the dictionary.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.result(sib.dict({1:1, 2:5/2, 3:100/3}))
{1:1, 2:5/2, 3:100/3}
```

```
sage: sib.result(sib.dict([('hello', 'sunshine'), ('goodbye', 'rain')]))
{'hello':'sunshine', 'goodbye':'rain'}
```

empty_subscript (*parent*)

Given a SageInputExpression representing foo, produces a SageInputExpression representing foo[]. Since this is not legal Python syntax, it is useful only for producing the sage generator syntax for a polynomial ring.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.result(sib.empty_subscript(sib(2) + sib(3)))
(2 + 3)[]
```

The following calls this method indirectly.:

```
sage: sage_input(polygen(ZZ['y']))
R.<x> = ZZ['y'][]
x
```

float_str (*n*)

Given a string representing a floating-point number, produces a SageInputExpression that formats as that string.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.result(sib.float_str(repr(RR(e))))
2.71828182845905
```

gen (*parent*, *n=0*)

Given a parent, returns a SageInputExpression for the n-th (default 0) generator of the parent.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.result(sib.gen(ZZ['y']))
R.<y> = ZZ[]
y
```

getattr (*sie*, *attr*)

Given a SageInputExpression representing foo and an attribute name bar, produce a SageInputExpression representing foo.bar. Normally, you could just use attribute-access syntax, but that doesn't work if bar is some attribute that bypasses __getattr__ (such as if bar is '__getattr__' itself).

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.getattr(ZZ, '__getattr__')
{getattr: {atomic:ZZ}.__getattr__}
sage: sib.getattr(sib.name('foo'), '__new__')
{getattr: {atomic:foo}.__new__}
```

id_cache(*x*, *sie*, *name*)

INPUT:

- `x` - an arbitrary value
- `sie` - a `SageInputExpression`
- `name` - a requested variable name

Enters `x` and `sie` in a cache, so that subsequent calls `self(x)` will directly return `sie`. Also, marks the requested name of this `sie` to be `name`. Differs from the method{cache} method in that the cache is keyed by `id(x)` instead of by `x`.

This may be called on values of an arbitrary type, which may be useful if the values are both large and likely to be used multiple times in a single expression; it should be preferred to method{cache} if equality on the values is difficult or impossible to compute.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: x = polygen(ZZ)
sage: sib = SageInputBuilder()
sage: my_42 = 42*x
sage: sie42 = sib(my_42)
sage: sib.id_cache(my_42, sie42, 'the_ultimate_answer')
sage: sib.result(sib(my_42) + sib(my_42))
R.<x> = ZZ[]
the_ultimate_answer = 42*x
the_ultimate_answer + the_ultimate_answer
```

Since id_cache keys off of object identity ("is"), the following does not trigger the cache.:

```
sage: sib.result(sib(42*x) + sib(42*x))
42*x + 42*x
```

Note that we don't assign the result to a variable if the value is only used once.:

```
sage: sib = SageInputBuilder()
sage: my_42 = 42*x
sage: sie42 = sib(my_42)
sage: sib.id_cache(my_42, sie42, 'the_ultimate_answer')
sage: sib.result(sib(my_42) + sib(43*x))
R.<x> = ZZ[]
42*x + 43*x
```

import_name(*module*, *name*, *alt_name=None*)

INPUT:

- `module`, `name`, `alt_name` – strings

Creates an expression that will import a name from a module and then use that name.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: v1 = sib.import_name('sage.foo.bar', 'baz')
sage: v2 = sib.import_name('sage.foo.bar', 'ZZ', 'not_the_real_ZZ')
sage: sib.result(v1+v2)
```

```
from sage.foo.bar import baz
from sage.foo.bar import ZZ as not_the_real_ZZ
baz + not_the_real_ZZ
```

We adjust the names if there is a conflict.:

```
sage: sib = SageInputBuilder()
sage: v1 = sib.import_name('sage.foo', 'poly')
sage: v2 = sib.import_name('sage.bar', 'poly')
sage: sib.result(v1+v2)
from sage.foo import poly as poly1
from sage.bar import poly as poly2
poly1 + poly2
```

int (*n*)

Return a raw SIE from the integer `n`

As it is raw, it may read back as a Sage Integer, a Python int or a Python long, depending on its size and whether the preparser is enabled.

INPUT:

- `n` - a Sage Integer, a Python int or a Python long

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.result(sib.int(-3^50))
-717897987691852588770249

sage: sib = SageInputBuilder()
sage: sib.result(sib.int(long(2^65)))
36893488147419103232

sage: sib = SageInputBuilder()
sage: sib.result(sib.int(-42r))
-42
```

name (*n*)

Given a string representing a Python name, produces a `SageInputExpression` for that name.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.result(sib.name('pi') + sib.name('e'))
pi + e
```

parent_with_gens (*parent, sie, gen_names, name, gens_syntax=None*)

This method is used for parents with generators, to manage the sage preparser generator syntax (like `K.<x> = QQ[]`).

The method{_sage_input_} method of a parent class with generators should construct a `SageInputExpression` for the parent, and then call this method with the parent itself, the constructed SIE, a sequence containing the names of the generators, and (optionally) another SIE to use if the sage generator syntax is used; typically this will be the same as the first SIE except omitting a `names` parameter.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder


sage: def test_setup(use_gens=True, preparse=True):
...       sib = SageInputBuilder(preparse=preparse)
...       gen_names=('foo', 'bar')
...       parent = "some parent"
...       normal_sie = sib.name('make_a_parent')(names=gen_names)
...       if use_gens:
...           gens_sie = sib.name('make_a_parent')()
...       else:
...           gens_sie = None
...       name = 'the_thing'
...       result = sib.parent_with_gens(parent, normal_sie,
...                                     gen_names, name,
...                                     gens_syntax=gens_sie)
...       return sib, result

sage: sib, par_sie = test_setup()
sage: sib.result(par_sie)
make_a_parent(names=('foo', 'bar'))

sage: sib, par_sie = test_setup()
sage: sib.result(sib(3) * sib.gen("some parent", 0))
the_thing.<foo,bar> = make_a_parent()
3*foo

sage: sib, par_sie = test_setup(preparse=False)
sage: sib.result(par_sie)
make_a_parent(names=('foo', 'bar'))

sage: sib, par_sie = test_setup(preparse=False)
sage: sib.result(sib(3) * sib.gen("some parent", 0))
the_thing = make_a_parent(names=('foo', 'bar'))
foo,bar = the_thing.gens()
ZZ(3)*foo

sage: sib, par_sie = test_setup(use_gens=False)
sage: sib.result(par_sie)
make_a_parent(names=('foo', 'bar'))

sage: sib, par_sie = test_setup(use_gens=False)
sage: sib.result(sib(3) * sib.gen("some parent", 0))
the_thing = make_a_parent(names=('foo', 'bar'))
foo,bar = the_thing.gens()
3*foo

sage: sib, par_sie = test_setup()
sage: sib.result(par_sie - sib.gen("some parent", 1))
the_thing.<foo,bar> = make_a_parent()
the_thing - bar
```

preparse()

Checks the preparse status.

It returns `True` if the preparser will be enabled, `False` if it will be disabled, and `None` if the result must work whether or not the preparser is enabled.

For example, this is useful in the method{_sage_input_} methods of `Integer` and `RealNumber`; but most method{_sage_input_} methods will not need to examine this.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder
sage: SageInputBuilder().preparse()
True
sage: SageInputBuilder(preparse=False).preparse()
False
```

prod (*factors*, *simplify=False*)

Given a sequence, returns a `SageInputExpression` for the product of the elements.

With `simplify=True`, performs some simplifications first. If any element is formatted as a string `'0'`, then that element is returned directly. If any element is formatted as a string `'1'`, then it is removed from the sequence (unless it is the only element in the sequence). And any negations are removed from the elements and moved to the outside of the product.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.result(sib.prod([-1, 0, 1, -2]))
-1*0*1*-2

sage: sib = SageInputBuilder()
sage: sib.result(sib.prod([-1, 0, 1, 2], simplify=True))
0

sage: sib = SageInputBuilder()
sage: sib.result(sib.prod([-1, 2, -3, -4], simplify=True))
-2*3*4

sage: sib = SageInputBuilder()
sage: sib.result(sib.prod([-1, 1, -1, -1], simplify=True))
-1

sage: sib = SageInputBuilder()
sage: sib.result(sib.prod([1, 1, 1], simplify=True))
1
```

result (*e*)

Given a `SageInputExpression` constructed using `self`, returns a tuple of a list of commands and an expression (and possibly a dictionary of local variables) suitable for `sage_eval()`.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: r = sib.result(sib(6) * sib(7)); r
6*7
sage: tuple(r)
('', '6*7')
```

share (*sie*)

Mark the given expression as sharable, so that it will be replaced by a variable if it occurs multiple times in the expression. (Most non-single-token expressions are already sharable.)

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder
```

Without explicitly using .share(), string literals are not shared:

```
sage: sib = SageInputBuilder()
sage: e = sib('hello')
sage: sib.result(sib((e, e)))
('hello', 'hello')
```

See the difference if we use .share():

```
sage: sib = SageInputBuilder()
sage: e = sib('hello')
sage: sib.share(e)
sage: sib.result(sib((e, e)))
si = 'hello'
(si, si)
```

sum(*terms*, *simplify=False*)

Given a sequence, returns a `SageInputExpression` for the product of the elements.

With `simplify=True`, performs some simplifications first. If any element is formatted as a string `'0'`, then it is removed from the sequence (unless it is the only element in the sequence); and any instances of `a + -b` are changed to `a - b`.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.result(sib.sum([-1, 0, 1, 0, -1]))
-1 + 0 + 1 + 0 + -1

sage: sib = SageInputBuilder()
sage: sib.result(sib.sum([-1, 0, 1, 0, -1], simplify=True))
-1 + 1 - 1

sage: sib = SageInputBuilder()
sage: sib.result(sib.sum([0, 0, 0], simplify=True))
0
```

use_variable(*sie*, *name*)

Marks the `SageInputExpression` `sie` to use a variable even if it is only referenced once. (If `sie` is the final top-level expression, though, it will not use a variable.)

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: e = sib.name('MatrixSpace')(ZZ, 10, 10)
sage: sib.use_variable(e, 'MS')
sage: sib.result(e.zero_matrix())
MS = MatrixSpace(ZZ, 10, 10)
MS.zero_matrix()
```

Without the call to use_variable, we get this instead:

```
sage: sib = SageInputBuilder()
sage: e = sib.name('MatrixSpace')(ZZ, 10, 10)
sage: sib.result(e.zero_matrix())
MatrixSpace(ZZ, 10, 10).zero_matrix()
```

And even with the call to use_variable, we don't use a variable here:

```
sage: sib = SageInputBuilder()
sage: e = sib.name('MatrixSpace')(ZZ, 10, 10)
sage: sib.use_variable(e, 'MS')
sage: sib.result(e)
MatrixSpace(ZZ, 10, 10)
```

class `sage.misc.sage_input.`**`SageInputExpression`**(*sib*)

Bases: `object`

Subclasses of this class represent expressions for `sage_input()`. sage classes should define a method{_sage_input_} method, which will return an instance of `SageInputExpression`, created using methods of `SageInputBuilder`.

To the extent possible, operations on `SageInputExpression` objects construct a new `SageInputExpression` representing that operation. That is, if `a` is a `SageInputExpression`, then `a + b` constructs a `SageInputExpression` representing this sum. This also works for attribute access, function calls, subscripts, etc. Since arbitrary attribute accesses might be used to construct a new attribute-access expression, all internal attributes and methods have names that begin with `_sie_` to reduce the chance of collisions.

It is expected that instances of this class will not be directly created outside this module; instead, instances will be created using methods of `SageInputBuilder` and `SageInputExpression`.

Values of type `SageInputExpression` print in a fairly ugly way, that reveals the internal structure of the expression tree.

class `sage.misc.sage_input.`**`SageInputFormatter`**

An instance of this class is used to keep track of variable names and a sequence of generated commands during the `sage_input()` formatting process.

`format`(*e*, *prec*)

Format a Sage input expression into a string.

INPUT:

- `e` - a `SageInputExpression`
- `prec` - an integer representing a precedence level

First, we check to see if `e` should be replaced by a variable. If so, we generate the command to assign the variable, and return the name of the variable.

Otherwise, we format the expression by calling its method{_sie_format} method, and add parentheses if necessary.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputBuilder, SageInputFormatter
```

```
sage: sib = SageInputBuilder()
sage: sif = SageInputFormatter()
sage: sie = sib(GF(5))
```

Here we `cheat` by calling method{_sie_prepare} twice, to make it use a variable.:

```
sage: sie._sie_prepare(sif)
sage: sie._sie_prepare(sif)
sage: sif._commands
''
sage: sif.format(sie, 0)
'GF_5'
sage: sif._commands
'GF_5 = GF(5)\n'
```

We demonstrate the use of commands, by showing how to construct code that will produce a random matrix:

```
sage: sib = SageInputBuilder()
sage: sif = SageInputFormatter()
sage: sie = sib.name('matrix')(sib.name('ZZ'), 10, 10)
sage: sib.command(sie, sie.randomize())
sage: sie._sie_prepare(sif)
sage: sif._commands
''
sage: sif.format(sie, 0)
'si'
sage: sif._commands
'si = matrix(ZZ, 10, 10)\nsi.randomize()\n'
```

get_name (*name*)

Return a name corresponding to a given requested name. If only one request for a name is received, then we will use the requested name; otherwise, we will add numbers to the end of the name to make it unique.

If the input name is `None`, then it is treated as a name of `'si'`.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputFormatter

sage: sif = SageInputFormatter()
sage: names = ('x', 'x', 'y', 'z')
sage: for n in names: sif.register_name(n)
sage: for n in names: sif.get_name(n)
'x1'
'x2'
'y'
'z'
```

register_name (*name*)

Register that some value would like to use a given name. If only one request for a name is received, then we will use the requested name; otherwise, we will add numbers to the end of the name to make it unique.

If the input name is `None`, then it is treated as a name of `'si'`.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputFormatter

sage: sif = SageInputFormatter()
sage: sif._names, sif._dup_names
(set(), {})
sage: sif.register_name('x')
sage: sif.register_name('y')
sage: sif._names, sif._dup_names
({'x', 'y'}, {})
```

```
sage: sif.register_name('x')
sage: sif._names, sif._dup_names
({'x', 'y'}, {'x': 0})
```

sage.misc.sage_input.**sage_input**(*x*, *preparse=True*, *verify=False*, *allow_locals=False*)

Return a sequence of commands that can be used to rebuild the object `x`.

INPUT:

- `x` - the value we want to find an input form for
- `preparse` - (default `True`) Whether to generate code that requires the preparser. With `True`, generated code requires the preparser. With `False`, generated code requires that the preparser not be used. With `None`, generated code will work whether or not the preparser is used.
- `verify` - (default `False`) If `True`, then the answer will be evaluated with `sage_eval()`, and an exception will be raised if the result is not equal to the original value. (In fact, for `verify=True`, `sage_input()` is effectively run three times, with `preparse` set to `True`, `False`, and `None`, and all three results are checked.) This is particularly useful for doctests.
- `allow_locals` - (default `False`) If `True`, then values that `sage_input()` cannot handle are returned in a dictionary, and the returned code assumes that this dictionary is passed as the `locals` parameter of `sage_eval()`. (Otherwise, if `sage_input()` cannot handle a value, an exception is raised.)

EXAMPLES:

```
sage: sage_input(GF(2)(1))
GF(2)(1)
sage: sage_input((GF(2)(0), GF(2)(1)), verify=True)
# Verified
GF_2 = GF(2)
(GF_2(0), GF_2(1))
```

When the preparser is enabled, we use the sage generator syntax.:

```
sage: K.<x> = GF(5)[]
sage: sage_input(x^3 + 2*x, verify=True)
# Verified
R.<x> = GF(5)[]
x^3 + 2*x
sage: sage_input(x^3 + 2*x, preparse=False)
R = GF(5)['x']
x = R.gen()
x**3 + 2*x
```

The result of `sage_input()` is actually a pair of strings with a special `__repr__` method to print nicely.:

```
sage: r = sage_input(RealField(20)(pi), verify=True)
sage: r
# Verified
RealField(20)(3.1415939)
sage: isinstance(r, tuple)
True
sage: len(r)
2
sage: tuple(r)
('# Verified\n', 'RealField(20)(3.1415939)')
```

We cannot find an input form for a function.:

```
sage: sage_input((3, lambda x: x))
Traceback (most recent call last):
...
ValueError: Can't convert <function <lambda> at 0x...> to sage_input form
```

But we can have `sage_input()` continue anyway, and return an input form for the rest of the expression, with `allow_locals=True`.:

```
sage: r = sage_input((3, lambda x: x), verify=True, allow_locals=True)
sage: r
LOCALS:
    _sil1: <function <lambda> at 0x...>
# Verified
(3, _sil1)
sage: tuple(r)
('# Verified\n', '(3, _sil1)', {'_sil1': <function <lambda> at 0x...>})
```

`sage.misc.sage_input.`**`verify_same`**(*a*, *b*)

Verify that two Sage values are the same. This is an extended equality test; it checks that the values are equal and that their parents are equal. (For values which are not Elements, the types are checked instead.)

If the values are the same, we return `None`; otherwise, we raise an exception.

EXAMPLES:

```
sage: from sage.misc.sage_input import verify_same
sage: verify_same(1, 1)
sage: verify_same(1, 2)
Traceback (most recent call last):
...
AssertionError: Expected 1 == 2
sage: verify_same(1, 1r)
Traceback (most recent call last):
...
AttributeError: 'int' object has no attribute 'parent'
sage: verify_same(1r, 1)
Traceback (most recent call last):
...
    assert(type(a) == type(b))
AssertionError
sage: verify_same(5, GF(7)(5))
Traceback (most recent call last):
...
    assert(a.parent() == b.parent())
AssertionError
```

`sage.misc.sage_input.`**`verify_si_answer`**(*x*, *answer*, *preparse*)

Verify that evaluating `answer` gives a value equal to `x` (with the same parent/type). If `preparse` is `True` or `False`, then we evaluate `answer` with the preparser enabled or disabled, respectively; if `preparse` is `None`, then we evaluate `answer` both with the preparser enabled and disabled and check both results.

On success, we return `None`; on failure, we raise an exception.

INPUT:

- `x` - an arbitrary Sage value
- `answer` - a string, or a `SageInputAnswer`
- `preparse` – `True`, `False`, or `None`

EXAMPLES:

```
sage: from sage.misc.sage_input import verify_si_answer
sage: verify_si_answer(1, '1', True)
sage: verify_si_answer(1, '1', False)
Traceback (most recent call last):
...
AttributeError: 'int' object has no attribute 'parent'
sage: verify_si_answer(1, 'ZZ(1)', None)
```

CHAPTER

THIRTYONE

RANDOM TESTING

Some Sage modules do random testing in their doctests; that is, they construct test cases using a random number generator. To get the broadest possible test coverage, we want everybody who runs the doctests to use a different random seed; but we also want to be able to reproduce the problems when debugging. This module provides a decorator to help write random testers that meet these goals.

sage.misc.random_testing.**random_testing**(*fn*)

This decorator helps create random testers. These can be run as part of the standard Sage test suite; everybody who runs the test will use a different random number seed, so many different random tests will eventually be run.

INPUT:

- `fn` - The function that we are wrapping for random testing.

The resulting function will take two additional arguments, *seed* (default `None`) and *print_seed* (default `False`). The result will set the random number seed to the given seed value (or to a truly random value, if *seed* is not specified), then call the original function. If *print_seed* is true, then the seed will be printed before calling the original function. If the original function raises an exception, then the random seed that was used will be displayed, along with a message entreating the user to submit a bug report. All other arguments will be passed through to the original function.

Here is a set of recommendations for using this wrapper.

The function to be tested should take arguments specifying the difficulty of the test (size of the test cases, number of iterations, etc.), as well as an argument *verbose* (defaulting to false). With *verbose* true, it should print the values being tested. Suppose `test_foo()` takes an argument for number of iterations. Then the doctests could be:

```
test_foo(2, verbose=True, seed=0)
test_foo(10)
test_foo(100) # long time
```

The first doctest, with the specified seed and `verbose=True`, simply verifies that the tests really are reproducible (that `test_foo` is correctly using the `randstate` framework). The next two tests use truly random seeds, and will print out the seed used if the test fails (raises an exception).

If you want a very long-running test using this setup, you should do something like:

```
for _ in xrange(10^10): test_foo(100)
```

instead of:

```
test_foo(10^12)
```

If the test fails after several hours, the latter snippet would make you rerun the test for several hours while reproducing and debugging the problem. With the former snippet, you only need to rerun `test_foo(100)` with a known-failing random seed.

See `sage.misc.random_testing.test_add_commutes()` for a simple example using this decorator, and `sage.rings.tests` for realistic uses.

Setting *print_seed* to true is useless in doctests, because the random seed printed will never match the expected doctest result (and using `# random` means the doctest framework will never report an error even if one happens). However, it is useful if you have a random test that sometimes segfaults. The normal print-the-random-seed-on-exceptions won't work then, so you can run:

```
while True: test_foo(print_seed=True)
```

and look at the last seed that was printed before it crashed.

TESTS:

```
sage: from sage.misc.random_testing import random_testing
sage: def foo(verbose=False):
...       'oh look, a docstring'
...       n = ZZ.random_element(2^50)
...       if verbose:
...           print "Random value: %s" % n
...       assert(n == 49681376900427)
sage: foo = random_testing(foo)
sage: foo(seed=0, verbose=True)
Random value: 49681376900427
sage: foo(seed=15, verbose=True)
Random value: 1049538412064764
Random testing has revealed a problem in foo
Please report this bug!  You may be the first
person in the world to have seen this problem.
Please include this random seed in your bug report:
Random seed: 15
AssertionError()
sage: foo() # random
Random testing has revealed a problem in foo
Please report this bug!  You may be the first
person in the world to have seen this problem.
Please include this random seed in your bug report:
Random seed: 272500700755151445506092479579811710040
AssertionError()
sage: foo.__doc__
'oh look, a docstring'
sage: foo.__name__
'foo'
sage: def bar(): pass
sage: bar = random_testing(bar)
sage: bar(print_seed=True) # random
Random seed: 262841091890156346923539765543814146051
```

`sage.misc.random_testing.`**`test_add_commutes`**`(*args, **kwargs)`

This is a simple demonstration of the `random_testing()` decorator and its recommended usage.

We test that addition is commutative over rationals.

EXAMPLES:

```
sage: from sage.misc.random_testing import test_add_commutes
sage: test_add_commutes(2, verbose=True, seed=0)
a == -4, b == 0 ...
Passes!
a == -1/2, b == -1/95 ...
Passes!
```

```
sage: test_add_commutes(10)
sage: test_add_commutes(1000) # long time
```

sage.misc.random_testing.**test_add_is_mul**(**args*, ***kwargs*)

This example demonstrates a failing `random_testing()` test, and shows how to reproduce the error.

DO NOT USE THIS AS AN EXAMPLE OF HOW TO USE `random_testing()`! Instead, look at `sage.misc.random_testing.test_add_commutes()`.

We test that `a+b == a*b`, for *a*, *b* rational. This is of course false, so the test will almost always fail.

EXAMPLES:

```
sage: from sage.misc.random_testing import test_add_is_mul
```

We start by testing that we get reproducible results when setting *seed* to 0.

```
sage: test_add_is_mul(2, verbose=True, seed=0)
a == -4, b == 0 ...
Random testing has revealed a problem in test_add_is_mul
Please report this bug!  You may be the first
person in the world to have seen this problem.
Please include this random seed in your bug report:
Random seed: 0
AssertionError()
```

Normally in a `@random_testing` doctest, we would leave off the `verbose=True` and the `# random`. We put it in here so that we can verify that we are seeing the exact same error when we reproduce the error below.

```
sage: test_add_is_mul(10, verbose=True) # random
a == -2/7, b == 1 ...
Random testing has revealed a problem in test_add_is_mul
Please report this bug!  You may be the first
person in the world to have seen this problem.
Please include this random seed in your bug report:
Random seed: 216390410596009428782506007128692114173
AssertionError()
```

OK, now assume that some user has reported a `test_add_is_mul()` failure. We can specify the same *random_seed* that was found in the bug report, and we will get the exact same failure so that we can debug the "problem".

```
sage: test_add_is_mul(10, verbose=True, seed=216390410596009428782506007128692114173)
a == -2/7, b == 1 ...
Random testing has revealed a problem in test_add_is_mul
Please report this bug!  You may be the first
person in the world to have seen this problem.
Please include this random seed in your bug report:
Random seed: 216390410596009428782506007128692114173
AssertionError()
```

CHAPTER

THIRTYTWO

INSPECT PYTHON, SAGE, AND CYTHON OBJECTS

This module extends parts of Python's inspect module to Cython objects.

AUTHORS:

- originally taken from Fernando Perez's IPython
- William Stein (extensive modifications)
- Nick Alexander (extensions)
- Nick Alexander (testing)
- Simon King (some extension for Cython, generalisation of SageArgSpecVisitor)

EXAMPLES:

```
sage: from sage.misc.sageinspect import *
```

Test introspection of modules defined in Python and Cython files:

Cython modules:

```
sage: sage_getfile(sage.rings.rational)
'.../rational.pyx'
```

```
sage: sage_getdoc(sage.rings.rational).lstrip()
'Rational Numbers...'
```

```
sage: sage_getsource(sage.rings.rational)[5:]
'Rational Numbers...'
```

Python modules:

```
sage: sage_getfile(sage.misc.sageinspect)
'.../sageinspect.py'
```

```
sage: print sage_getdoc(sage.misc.sageinspect).lstrip()[:40]
Inspect Python, Sage, and Cython objects
```

```
sage: sage_getsource(sage.misc.sageinspect).lstrip()[5:-1]
'Inspect Python, Sage, and Cython objects...'
```

Test introspection of classes defined in Python and Cython files:

Cython classes:

```
sage: sage_getfile(sage.rings.rational.Rational)
'.../rational.pyx'
```

```
sage: sage_getdoc(sage.rings.rational.Rational).lstrip()
'A rational number...'
```

```
sage: sage_getsource(sage.rings.rational.Rational)
'cdef class Rational...'
```

Python classes:

```
sage: sage_getfile(BlockFinder)
'.../sage/misc/sageinspect.py'
```

```
sage: sage_getdoc(BlockFinder).lstrip()
'Provide a tokeneater() method to detect the...'
```

```
sage: sage_getsource(BlockFinder)
'class BlockFinder:...'
```

Python classes with no docstring, but an __init__ docstring:

```
sage: class Foo:
...     def __init__(self):
...         'docstring'
...         pass
...
sage: sage_getdoc(Foo)
'docstring\n'
```

Test introspection of functions defined in Python and Cython files:

Cython functions:

```
sage: sage_getdef(sage.rings.rational.make_rational, obj_name='mr')
'mr(s)'
```

```
sage: sage_getfile(sage.rings.rational.make_rational)
'.../rational.pyx'
```

```
sage: sage_getdoc(sage.rings.rational.make_rational).lstrip()
'Make a rational number ...'
```

```
sage: sage_getsource(sage.rings.rational.make_rational)[4:]
'make_rational(s):...'
```

Python functions:

```
sage: sage_getdef(sage.misc.sageinspect.sage_getfile, obj_name='sage_getfile')
'sage_getfile(obj)'
```

```
sage: sage_getfile(sage.misc.sageinspect.sage_getfile)
'.../sageinspect.py'
```

```
sage: sage_getdoc(sage.misc.sageinspect.sage_getfile).lstrip()
'Get the full file name associated to "obj" as a string...'
```

```
sage: sage_getsource(sage.misc.sageinspect.sage_getfile)[4:]
'sage_getfile(obj):...'
```

Unfortunately, there is no argspec extractable from builtins:

```
sage: sage_getdef(''.find, 'find')
'find( [noargspec] )'

sage: sage_getdef(str.find, 'find')
'find( [noargspec] )'
```

By trac ticket #9976 and trac ticket #14017, introspection also works for interactively defined Cython code, and with rather tricky argument lines:

```
sage: cython('def foo(unsigned int x=1, a=\')"\', b={not (2+1==3):\'bar\'}, *args, **kwds): return')
sage: print sage_getsource(foo)
def foo(unsigned int x=1, a=')"', b={not (2+1==3):'bar'}, *args, **kwds): return
sage: sage_getargspec(foo)
ArgSpec(args=['x', 'a', 'b'], varargs='args', keywords='kwds', defaults=(1, ')"', {False: 'bar'}))
```

class sage.misc.sageinspect.**BlockFinder**

Provide a tokeneater() method to detect the end of a code block.

This is the Python library's `inspect.BlockFinder` modified to recognize Cython definitions.

tokeneater(*type*, *token*, *srow_scol*, *erow_ecol*, *line*)

x.__init__(...) initializes x; see help(type(x)) for signature

class sage.misc.sageinspect.**SageArgSpecVisitor**

Bases: `ast.NodeVisitor`

A simple visitor class that walks an abstract-syntax tree (AST) for a Python function's argspec. It returns the contents of nodes representing the basic Python types: None, booleans, numbers, strings, lists, tuples, and dictionaries. We use this class in `_sage_getargspec_from_ast()` to extract an argspec from a function's or method's source code.

EXAMPLES:

```
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: visitor.visit(ast.parse('[1,2,3]').body[0].value)
[1, 2, 3]
sage: visitor.visit(ast.parse("{'a':('e',2,[None,({False:True},'pi')]), 37.0:'temp'}").body[0].v
{37.0: 'temp', 'a': ('e', 2, [None, ({False: True}, 'pi')])}
sage: v = ast.parse("jc = ['veni', 'vidi', 'vici']").body[0]; v
<_ast.Assign object at ...>
sage: [x for x in dir(v) if not x.startswith('__')]
['_attributes', '_fields', 'col_offset', 'lineno', 'targets', 'value']
sage: visitor.visit(v.targets[0])
'jc'
sage: visitor.visit(v.value)
['veni', 'vidi', 'vici']
```

visit_BinOp(*node*)

Visit a Python AST `ast.BinOp` node.

INPUT:

- `node` - the node instance to visit

OUTPUT:

- The result that `node` represents

AUTHOR:

- Simon King

EXAMPLES:

```
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit(ast.parse(x).body[0].value)
sage: [vis(d) for d in ['(3+(2*4))', '7|8', '5^3', '7/3', '7//3', '3<<4']] #indirect doctest
[11, 15, 6, 2, 2, 48]
```

visit_BoolOp(*node*)

Visit a Python AST `ast.BoolOp` node.

INPUT:

- `node` - the node instance to visit

OUTPUT:

- The result that `node` represents

AUTHOR:

- Simon King

EXAMPLES:

```
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit(ast.parse(x).body[0].value)
sage: [vis(d) for d in ['True and 1', 'False or 3 or None', '3 and 4']] #indirect doctest
[1, 3, 4]
```

visit_Compare(*node*)

Visit a Python AST `ast.Compare` node.

INPUT:

- `node` - the node instance to visit

OUTPUT:

- The result that `node` represents

AUTHOR:

- Simon King

EXAMPLES:

```
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_Compare(ast.parse(x).body[0].value)
sage: [vis(d) for d in ['1<2==2!=3', '1==1>2', '1<2>1', '1<3<2<4']]
[True, False, True, False]
```

visit_Dict(*node*)

Visit a Python AST `ast.Dict` node.

INPUT:

- `node` - the node instance to visit

OUTPUT:

•the dictionary the `node` represents

EXAMPLES:

```
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_Dict(ast.parse(x).body[0].value)
sage: [vis(d) for d in ['{}', "{1:one, 'two':2, other:bother}"]]
[{}, {1: 'one', 'other': 'bother', 'two': 2}]
```

visit_List (*node*)

Visit a Python AST `ast.List` node.

INPUT:

•`node` - the node instance to visit

OUTPUT:

•the list the `node` represents

EXAMPLES:

```
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_List(ast.parse(x).body[0].value)
sage: [vis(l) for l in ['[]', "['s', 't', 'u']", '[[e], [], [pi]]']]
[[], ['s', 't', 'u'], [['e'], [], ['pi']]]
```

visit_Name (*node*)

Visit a Python AST `ast.Name` node.

INPUT:

•`node` - the node instance to visit

OUTPUT:

•None, True, False, or the `node`'s name as a string.

EXAMPLES:

```
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_Name(ast.parse(x).body[0].value)
sage: [vis(n) for n in ['True', 'False', 'None', 'foo', 'bar']]
[True, False, None, 'foo', 'bar']
sage: [type(vis(n)) for n in ['True', 'False', 'None', 'foo', 'bar']]
[<type 'bool'>, <type 'bool'>, <type 'NoneType'>, <type 'str'>, <type 'str'>]
```

visit_Num (*node*)

Visit a Python AST `ast.Num` node.

INPUT:

•`node` - the node instance to visit

OUTPUT:

•the number the `node` represents

EXAMPLES:

```
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_Num(ast.parse(x).body[0].value)
```

```
sage: [vis(n) for n in ['123', '0.0', str(-pi.n())]]
[123, 0.0, -3.14159265358979]
```

visit_Str(*node*)

Visit a Python AST `ast.Str` node.

INPUT:

- `node` - the node instance to visit

OUTPUT:

- the string the `node` represents

EXAMPLES:

```
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_Str(ast.parse(x).body[0].value)
sage: [vis(s) for s in ['"abstract"', "u'syntax'", '''r"tr\ee"''']]
['abstract', u'syntax', 'tr\\ee']
```

visit_Tuple(*node*)

Visit a Python AST `ast.Tuple` node.

INPUT:

- `node` - the node instance to visit

OUTPUT:

- the tuple the `node` represents

EXAMPLES:

```
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_Tuple(ast.parse(x).body[0].value)
sage: [vis(t) for t in ['()', '(x,y)', '("Au", "Al", "Cu")']]
[(), ('x', 'y'), ('Au', 'Al', 'Cu')]
```

visit_UnaryOp(*node*)

Visit a Python AST `ast.BinOp` node.

INPUT:

- `node` - the node instance to visit

OUTPUT:

- The result that `node` represents

AUTHOR:

- Simon King

EXAMPLES:

```
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_UnaryOp(ast.parse(x).body[0].value)
sage: [vis(d) for d in ['+(3*2)', '-(3*2)']]
[6, -6]
```

sage.misc.sageinspect.**isclassinstance**(*obj*)

Checks if argument is instance of non built-in class

INPUT: obj - object

EXAMPLES:

```
sage: from sage.misc.sageinspect import isclassinstance
sage: isclassinstance(int)
False
sage: isclassinstance(FreeModule)
True
sage: class myclass: pass
sage: isclassinstance(myclass)
False
sage: class mymetaclass(type): pass
sage: class myclass2:
...       __metaclass__ = mymetaclass
sage: isclassinstance(myclass2)
False
```

sage.misc.sageinspect.**sage_getargspec**(*obj*)

Return the names and default values of a function's arguments.

INPUT:

obj, any callable object

OUTPUT:

An ArgSpec is returned. This is a named tuple (args, varargs, keywords, defaults).

- args is a list of the argument names (it may contain nested lists).
- varargs and keywords are the names of the * and ** arguments or None.
- defaults is an n-tuple of the default values of the last n arguments.

NOTE:

If the object has a method _sage_argspec_ then the output of that method is transformed into a named tuple and then returned.

If a class instance has a method _sage_src_ then its output is studied to determine the argspec. This is because currently the CachedMethod decorator has no _sage_argspec_ method.

EXAMPLES:

```
sage: from sage.misc.sageinspect import sage_getargspec
sage: def f(x, y, z=1, t=2, *args, **keywords):
...       pass
sage: sage_getargspec(f)
ArgSpec(args=['x', 'y', 'z', 't'], varargs='args', keywords='keywords', defaults=(1, 2))
```

We now run sage_getargspec on some functions from the Sage library:

```
sage: sage_getargspec(identity_matrix)
ArgSpec(args=['ring', 'n', 'sparse'], varargs=None, keywords=None, defaults=(0, False))
sage: sage_getargspec(factor)
ArgSpec(args=['n', 'proof', 'int_', 'algorithm', 'verbose'], varargs=None, keywords='kwds', defa
```

In the case of a class or a class instance, the ArgSpec of the __new__, __init__ or __call__ method is returned:

```
sage: P.<x,y> = QQ[]
sage: sage_getargspec(P)
ArgSpec(args=['self', 'x'], varargs='args', keywords='kwds', defaults=(0,))
sage: sage_getargspec(P.__class__)
ArgSpec(args=['self', 'x'], varargs='args', keywords='kwds', defaults=(0,))
```

The following tests against various bugs that were fixed in trac ticket #9976:

```
sage: from sage.rings.polynomial.real_roots import bernstein_polynomial_factory_ratlist
sage: sage_getargspec(bernstein_polynomial_factory_ratlist.coeffs_bitsize)
ArgSpec(args=['self'], varargs=None, keywords=None, defaults=None)
sage: from sage.rings.polynomial.pbori import BooleanMonomialMonoid
sage: sage_getargspec(BooleanMonomialMonoid.gen)
ArgSpec(args=['self', 'i'], varargs=None, keywords=None, defaults=(0,))
sage: I = P*[x,y]
sage: sage_getargspec(I.groebner_basis)
ArgSpec(args=['self', 'algorithm', 'deg_bound', 'mult_bound', 'prot'],
varargs='args', keywords='kwds', defaults=('', None, None, False))
sage: cython("cpdef int foo(x,y) except -1: return 1")
sage: sage_getargspec(foo)
ArgSpec(args=['x', 'y'], varargs=None, keywords=None, defaults=None)
```

If a `functools.partial` instance is involved, we see no other meaningful solution than to return the argspec of the underlying function:

```
sage: def f(a,b,c,d=1): return a+b+c+d
...
sage: import functools
sage: f1 = functools.partial(f, 1,c=2)
sage: sage_getargspec(f1)
ArgSpec(args=['a', 'b', 'c', 'd'], varargs=None, keywords=None, defaults=(1,))
```

TESTS:

By trac ticket #9976, rather complicated cases work. In the following example, we dynamically create an extension class that returns some source code, and the example shows that the source code is taken for granted, i.e., the argspec of an instance of that class does not coincide with the argspec of its call method. That behaviour is intended, since a decorated method appears to have the generic signature `*args,**kwds`, but in fact it is only supposed to be called with the arguments requested by the underlying undecorated method. We saw an easy example above, namely `I.groebner_basis`. Here is a more difficult one:

```
sage: cython_code = [
... 'cdef class MyClass:',
... '    def _sage_src_(self):',
... '        return "def foo(x, a=\\\')\\\"\\\', b={(2+1):\\\'bar\\\', not 1:3, 3<<4:5}): return
... '    def __call__(self, m,n): return "something"']
sage: cython('\n'.join(cython_code))
sage: O = MyClass()
sage: print sage.misc.sageinspect.sage_getsource(O)
def foo(x, a=')"', b={(2+1):'bar', not 1:3, 3<<4:5}): return
sage: sage.misc.sageinspect.sage_getargspec(O)
ArgSpec(args=['x', 'a', 'b'], varargs=None, keywords=None, defaults=(')"', {False: 3, 48: 5, 3:
sage: sage.misc.sageinspect.sage_getargspec(O.__call__)
ArgSpec(args=['self', 'm', 'n'], varargs=None, keywords=None, defaults=None)

sage: cython('def foo(x, a=\'\\\')"\', b={not (2+1==3):\'bar\'}): return')
sage: print sage.misc.sageinspect.sage_getsource(foo)
def foo(x, a='\')"', b={not (2+1==3):'bar'}): return
```

```
sage: sage.misc.sageinspect.sage_getargspec(foo)
ArgSpec(args=['x', 'a', 'b'], varargs=None, keywords=None, defaults=('\')"', {False: 'bar'}))
```

The following produced a syntax error before the patch at trac ticket #11913:

```
sage: sage.misc.sageinspect.sage_getargspec(r.lm)
```

The following was fixed in trac ticket #16309:

```
sage: cython('''
....: class Foo:
....:     @staticmethod
....:     def join(categories, bint as_list = False, tuple ignore_axioms=(), tuple axioms=()): p
....: cdef class Bar:
....:     @staticmethod
....:     def join(categories, bint as_list = False, tuple ignore_axioms=(), tuple axioms=()): p
....:     cpdef meet(categories, bint as_list = False, tuple ignore_axioms=(), tuple axioms=()):
....: ''')
sage: sage_getargspec(Foo.join)
ArgSpec(args=['categories', 'as_list', 'ignore_axioms', 'axioms'], varargs=None, keywords=None,
sage: sage_getargspec(Bar.join)
ArgSpec(args=['categories', 'as_list', 'ignore_axioms', 'axioms'], varargs=None, keywords=None,
sage: sage_getargspec(Bar.meet)
ArgSpec(args=['categories', 'as_list', 'ignore_axioms', 'axioms'], varargs=None, keywords=None,
```

Test that trac ticket #17009 is fixed:

```
sage: sage_getargspec(gap)
ArgSpec(args=['self', 'x', 'name'], varargs=None, keywords=None, defaults=(None,))
```

AUTHORS:

- William Stein: a modified version of inspect.getargspec from the Python Standard Library, which was taken from IPython for use in Sage.
- Extensions by Nick Alexander
- Simon King: Return an `ArgSpec`, fix some bugs.

`sage.misc.sageinspect.`**`sage_getdef`**(*obj*, *obj_name=''*)

Return the definition header for any callable object.

INPUT:

- `obj` - function
- `obj_name` - string (optional, default '')

`obj_name` is prepended to the output.

EXAMPLES:

```
sage: from sage.misc.sageinspect import sage_getdef
sage: sage_getdef(identity_matrix)
'(ring, n=0, sparse=False)'
sage: sage_getdef(identity_matrix, 'identity_matrix')
'identity_matrix(ring, n=0, sparse=False)'
```

Check that trac ticket #6848 has been fixed:

```
sage: sage_getdef(RDF.random_element)
'(min=-1, max=1)'
```

If an exception is generated, None is returned instead and the exception is suppressed.

AUTHORS:

- William Stein
- extensions by Nick Alexander

sage.misc.sageinspect.**sage_getdoc**(*obj*, *obj_name=''*, *embedded_override=False*)

Return the docstring associated to `obj` as a string.

INPUT: `obj`, a function, module, etc.: something with a docstring.

If `obj` is a Cython object with an embedded position in its docstring, the embedded position is stripped.

If optional argument `embedded_override` is False (its default value), then the string is formatted according to the value of EMBEDDED_MODE. If this argument is True, then it is formatted as if EMBEDDED_MODE were True.

EXAMPLES:

```
sage: from sage.misc.sageinspect import sage_getdoc
sage: sage_getdoc(identity_matrix)[87:124]
'Return the n x n identity matrix over'
sage: def f(a,b,c,d=1): return a+b+c+d
...
sage: import functools
sage: f1 = functools.partial(f, 1,c=2)
sage: f.__doc__ = "original documentation"
sage: f1.__doc__ = "specialised documentation"
sage: sage_getdoc(f)
'original documentation\n'
sage: sage_getdoc(f1)
'specialised documentation\n'
```

AUTHORS:

- William Stein
- extensions by Nick Alexander

sage.misc.sageinspect.**sage_getfile**(*obj*)

Get the full file name associated to `obj` as a string.

INPUT: `obj`, a Sage object, module, etc.

EXAMPLES:

```
sage: from sage.misc.sageinspect import sage_getfile
sage: sage_getfile(sage.rings.rational)[-23:]
'sage/rings/rational.pyx'
sage: sage_getfile(Sq)[-42:]
'sage/algebras/steenrod/steenrod_algebra.py'
```

The following tests against some bugs fixed in trac ticket #9976:

```
sage: obj = sage.combinat.partition_algebra.SetPartitionsAk
sage: obj = sage.combinat.partition_algebra.SetPartitionsAk
sage: sage_getfile(obj)
'...sage/combinat/partition_algebra.py'
```

And here is another bug, fixed in trac ticket #11298:

```
sage: P.<x,y> = QQ[]
sage: sage_getfile(P)
'...sage/rings/polynomial/multi_polynomial_libsingular.pyx'
```

A problem fixed in trac ticket #16309:

```
sage: cython('''
....: class Bar: pass
....: cdef class Foo: pass
....: ''')
sage: sage_getfile(Bar)
'...pyx'
sage: sage_getfile(Foo)
'...pyx'
```

AUTHORS:

- Nick Alexander
- Simon King

`sage.misc.sageinspect.`**`sage_getsource`**`(`*obj*`)`

Return the source code associated to obj as a string, or None.

INPUT:

- `obj` – function, etc.

EXAMPLES:

```
sage: from sage.misc.sageinspect import sage_getsource
sage: sage_getsource(identity_matrix)[19:60]
'identity_matrix(ring, n=0, sparse=False):'
sage: sage_getsource(identity_matrix)[19:60]
'identity_matrix(ring, n=0, sparse=False):'
```

AUTHORS:

- William Stein
- extensions by Nick Alexander

`sage.misc.sageinspect.`**`sage_getsourcelines`**`(`*obj*`)`

Return a pair ([source_lines], starting line number) of the source code associated to obj, or None.

INPUT:

- `obj` – function, etc.

OUTPUT:

(source_lines, lineno) or None: `source_lines` is a list of strings, and `lineno` is an integer.

EXAMPLES:

```
sage: from sage.misc.sageinspect import sage_getsourcelines
sage: sage_getsourcelines(matrix)[1]
732
sage: sage_getsourcelines(matrix)[0][0][6:]
'MatrixFactory(object):\n'
```

TESTS:

```
sage: cython('''cpdef test_funct(x,y): return''')
sage: sage_getsourcelines(test_funct)
(['cpdef test_funct(x,y): return\n'], 6)
```

The following tests that an instance of `functools.partial` is correctly dealt with (see trac ticket #9976):

```
sage: obj = sage.combinat.partition_algebra.SetPartitionsAk
sage: sage_getsourcelines(obj)
(['def create_set_partition_function(letter, k):\n',
...
'    raise ValueError("k must be an integer or an integer + 1/2")\n'], 34)
```

Here are some cases that were covered in :trac‘11298‘; note that line numbers may easily change, and therefore we do not test them:

```
sage: P.<x,y> = QQ[]
sage: I = P*[x,y]
sage: sage_getsourcelines(P)
(['cdef class MPolynomialRing_libsingular(MPolynomialRing_generic):\n',
  '\n',
  '    def __cinit__(self):\n',
...)
sage: sage_getsourcelines(I)
(['class MPolynomialIdeal( MPolynomialIdeal_singular_repr, \\\n',
...)
sage: x = var('x')
sage: sage_getsourcelines(x)
(['cdef class Expression(CommutativeRingElement):\n',
  '    cpdef object pyobject(self):\n',
...)
sage: sage_getsourcelines(x)[0][-1]    # last line
'        return S\n'
```

We show some enhancements provided by trac ticket #11768. First, we use a dummy parent class that has defined an element class by a nested class definition:

```
sage: from sage.misc.nested_class_test import TestNestedParent
sage: from sage.misc.sageinspect import sage_getsource
sage: P = TestNestedParent()
sage: E = P.element_class
sage: E.__bases__
(<class sage.misc.nested_class_test.TestNestedParent.Element at ...>,
 <class 'sage.categories.sets_cat.Sets.element_class'>)
sage: print sage_getsource(E)
    class Element:
        "This is a dummy element class"
        pass
sage: print sage_getsource(P)
class TestNestedParent(UniqueRepresentation, Parent):
    ...
    class Element:
        "This is a dummy element class"
        pass
```

Here is another example that relies on a nested class definition in the background:

```
sage: C = AdditiveMagmas()
sage: HC = C.Homsets()
sage: sage_getsourcelines(HC)
```

```
(['    class Homsets(HomsetsCategory):\n', ...], ...)
```

Testing against a bug that has occured during work on #11768:

```
sage: P.<x,y> = QQ[]
sage: I = P*[x,y]
sage: sage_getsourcelines(I)
(['class MPolynomialIdeal( MPolynomialIdeal_singular_repr, \\\n',
  '                        MPolynomialIdeal_macaulay2_repr, \\\n',
  '                        MPolynomialIdeal_magma_repr, \\\n',
  '                        Ideal_generic ):\n',
  '    def __init__(self, ring, gens, coerce=True):\n',
  ...)
```

AUTHORS:

- William Stein
- Extensions by Nick Alexander
- Extension to interactive Cython code by Simon King
- Simon King: If a class has no docstring then let the class definition be found starting from the `__init__` method.
- Simon King: Get source lines for dynamic classes.

`sage.misc.sageinspect.`**`sage_getvariablename`**(*self*, *omit_underscore_names=True*)

Attempt to get the name of a Sage object.

INPUT:

- `self` – any object.
- `omit_underscore_names` – boolean, default `True`.

OUTPUT:

If the user has assigned an object `obj` to a variable name, then return that variable name. If several variables point to `obj`, return a sorted list of those names. If `omit_underscore_names` is True (the default) then omit names starting with an underscore "_".

This is a modified version of code taken from http://pythonic.pocoo.org/2009/5/30/finding-objects-names, written by Georg Brandl.

EXAMPLES:

```
sage: from sage.misc.sageinspect import sage_getvariablename
sage: A = random_matrix(ZZ, 100)
sage: sage_getvariablename(A)
'A'
sage: B = A
sage: sage_getvariablename(A)
['A', 'B']
```

If an object is not assigned to a variable, an empty list is returned:

```
sage: sage_getvariablename(random_matrix(ZZ, 60))
[]
```

CHAPTER

THIRTYTHREE

FORMAT SAGE DOCUMENTATION FOR VIEWING WITH IPYTHON AND THE NOTEBOOK

AUTHORS:

- William Stein (2005): initial version.
- Nick Alexander (2007): nodetex functions
- Nick Alexander (2008): search_src, search_def improvements
- Martin Albrecht (2008-03-21): parse LaTeX description environments in sagedoc
- John Palmieri (2009-04-11): fix for #5754 plus doctests
- Dan Drake (2009-05-21): refactor search_* functions, use system 'find' instead of sage -grep
- John Palmieri (2009-06-28): don't use 'find' – use Python (os.walk, re.search) instead.
- Simon King (2011-09): Use os.linesep, avoid destruction of embedding information, enable nodetex in a docstring. Consequently use sage_getdoc.

TESTS:

Check that argspecs of extension function/methods appear correctly, see trac ticket #12849:

```
sage: docfilename = os.path.join(SAGE_DOC, 'output', 'html', 'en', 'reference', 'calculus', 'sage', '
sage: for line in open(docfilename):
...       if "#sage.symbolic.expression.Expression.N" in line:
...           print line
<tt class="descname">N</tt><big>(</big><em>prec=None</em>, <em>digits=None</em>, <em>algorithm=None</
```

sage.misc.sagedoc.**constructions**

Sage constructions. Attempts to answer the question "How do I construct ... in Sage?"

EXAMPLES:

```
sage: constructions()   # indirect doctest, not tested
```

sage.misc.sagedoc.**detex**(*s*, *embedded=False*)

This strips LaTeX commands from a string; it is used by the `format` function to process docstrings for display from the command line interface.

INPUT:

- `s` - string
- `embedded` - boolean (optional, default False)

If `embedded` is False, then do the replacements in both `math_substitutes` and `nonmath_substitutes`. If True, then only do `nonmath_substitutes`.

OUTPUT:

string

EXAMPLES:

```
sage: from sage.misc.sagedoc import detex
sage: detex(r'Some math: `n \geq k`.  A website: \url{sagemath.org}.')
'Some math: n >= k.  A website: sagemath.org.\n'
sage: detex(r'More math: `x \mapsto y`.  {\bf Bold face}.')
'More math: x  |-->  y.  { Bold face}.\n'
sage: detex(r'`a, b, c, \ldots, z`')
'a, b, c, ..., z\n'
sage: detex(r'`a, b, c, \ldots, z`', embedded=True)
'`a, b, c, \\ldots, z`'
sage: detex(r'`\left(\lvert x\ast y \rvert\right]`')
'(| x * y |]\n'
sage: detex(r'`\left(\leq\le\leftarrow \rightarrow\to`')
'(<=<=leftarrow rightarrow-->\n'
```

sage.misc.sagedoc.**developer**

The Sage developer's guide. Learn to develop programs for Sage.

EXAMPLES:

```
sage: developer()  # indirect doctest, not tested
```

sage.misc.sagedoc.**format**(*s*, *embedded=False*)

noreplace Format Sage documentation `s` for viewing with IPython.

This calls `detex` on `s` to convert LaTeX commands to plain text, unless the directive `nodetex` is given in the first line of the string.

Also, if `s` contains a string of the form `<<<obj>>>`, then it replaces it with the docstring for `obj`, unless the directive `noreplace` is given in the first line. If an error occurs under the attempt to find the docstring for `obj`, then the substring `<<<obj>>>` is preserved.

Directives must be separated by a comma.

NOTE:

If the first line of the string provides embedding information, which is the case for doc strings from extension modules, then the first line will not be changed.

INPUT:

- `s` - string
- `embedded` - boolean (optional, default False)

OUTPUT: string

Set `embedded` equal to True if formatting for use in the notebook; this just gets passed as an argument to `detex`.

EXAMPLES:

```
sage: from sage.misc.sagedoc import format
sage: identity_matrix(2).rook_vector.__doc__[110:182]
'Let `A` be an `m` by `n` (0,1)-matrix. We identify `A` with a chessboard'

sage: format(identity_matrix(2).rook_vector.__doc__[110:182])
'Let A be an m by n (0,1)-matrix. We identify A with a chessboard\n'
```

If the first line of the string is 'nodetex', remove 'nodetex' but don't modify any TeX commands:

```
sage: format("nodetex\n`x \\geq y`")
'`x \\geq y`'
```

Testing a string enclosed in triple angle brackets:

```
sage: format('<<<identity_matrix')
'<<<identity_matrix\n'
sage: format('identity_matrix>>>')
'identity_matrix>>>\n'
sage: format('<<<identity_matrix>>>')[:28]
'Definition: identity_matrix('
```

TESTS:

We check that the todo Sphinx extension is correctly activated:

```
sage: sage.misc.sagedoc.format(sage.combinat.ranker.on_fly.__doc__)
"   Returns ...  Todo: add tests as in combinat::rankers\n"
```

We check that the embedding information of a doc string from an extension module is preserved, even if it is longer than a usual line. Moreover, a `nodetex` directive in the first "essential" line of the doc string is recognised. That has been implemented in trac ticket #11815:

```
sage: r = 'File: _local_user_with_a_very_long_name_that_would_normally_be_wrapped_sage_temp_mach
sage: print format(r)
File: _local_user_with_a_very_long_name_that_would_normally_be_wrapped_sage_temp_machine_name_12

some doc for a cython method
`x \geq y`
```

In the following use case, the `nodetex` directive would have been ignored prior to #11815:

```
sage: cython_code = ["def testfunc(x):",
... "    '''",
... "    nodetex",
... "    This is a doc string with raw latex",
... "",
... "    `x \\geq y`",
... "    '''",
... "    return -x"]
sage: cython('\n'.join(cython_code))
sage: from sage.misc.sageinspect import sage_getdoc
sage: print sage_getdoc(testfunc)

    This is a doc string with raw latex

    `x \geq y`
```

We check that the `noreplace` directive works, even combined with `nodetex` and an embedding information (see trac ticket #11817):

```
sage: print format('File: bla.py (starting at line 1)\nnodetex, noreplace\n<<<identity_matrix>>>
File: bla.py (starting at line 1)
<<<identity_matrix>>>`\not= 0`
```

If replacement is impossible, then no error is raised:

```
sage: print format('<<<bla\n<<<bla>>>\n<<<identity_matrix>>>')
<<<bla <<<bla>>>
```

```
Definition: identity_matrix(ring, n=0, sparse=False)

This function is available as identity_matrix(...) and
matrix.identity(...).

   Return the n x n identity matrix over the given ring.
...
```

sage.misc.sagedoc.**format_search_as_html**(*what*, *r*, *search*)

Format the output from search_src, search_def, or search_doc as html, for use in the notebook.

INPUT:

- what - (string) what was searched (source code or documentation)
- r - (string) the results of the search
- search - (string) what was being searched for

This function parses r: it should have the form FILENAME: string where FILENAME is the file in which the string that matched the search was found. Everything following the first colon is ignored; we just use the filename. If FILENAME ends in '.html', then this is part of the documentation; otherwise, it is in the source code. In either case, an appropriate link is created.

EXAMPLES:

```
sage: from sage.misc.sagedoc import format_search_as_html
sage: format_search_as_html('Source', 'algebras/steenrod_algebra_element.py:        an antihomom
'<html><font color="black"><h2>Search Source: antipode antihomomorphism</h2></font><font color="
sage: format_search_as_html('Other', 'html/en/reference/sage/algebras/steenrod_algebra_element.h
'<html><font color="black"><h2>Search Other: antipode antihomomorphism</h2></font><font color="d
```

sage.misc.sagedoc.**format_src**(*s*)

Format Sage source code s for viewing with IPython.

If s contains a string of the form "<<<obj>>>", then it replaces it with the source code for "obj".

INPUT: s - string

OUTPUT: string

EXAMPLES:

```
sage: from sage.misc.sagedoc import format_src
sage: format_src('unladen swallow')
'unladen swallow'
sage: format_src('<<<Sq>>>')[5:15]
'Sq(*nums):'
```

sage.misc.sagedoc.**help**(*module=None*)

If there is an argument module, print the Python help message for module. With no argument, print a help message about getting help in Sage.

EXAMPLES:

```
sage: help()
Welcome to Sage ...
```

sage.misc.sagedoc.**manual**

The Sage reference manual.

EXAMPLES:

```
sage: reference() # indirect doctest, not tested
sage: manual() # indirect doctest, not tested
```

sage.misc.sagedoc.**my_getsource**(*obj*, *oname*='')

Retrieve the source code for `obj`.

INPUT:

- `obj` – a Sage object, function, etc.
- `oname` – str (optional). A name under which the object is known. Currently ignored by Sage.

OUTPUT:

Its documentation (string)

EXAMPLES:

```
sage: from sage.misc.sagedoc import my_getsource
sage: s = my_getsource(identity_matrix)
sage: s[15:34]
'def identity_matrix'
```

sage.misc.sagedoc.**process_dollars**(*s*)

Replace dollar signs with backticks.

More precisely, do a regular expression search. Replace a plain dollar sign ($) by a backtick (`). Replace an escaped dollar sign (\$) by a dollar sign ($). Don't change a dollar sign preceded or followed by a backtick (`$ or $`), because of strings like "`$HOME`". Don't make any changes on lines starting with more spaces than the first nonempty line in `s`, because those are indented and hence part of a block of code or examples.

This also doesn't replaces dollar signs enclosed in curly braces, to avoid nested math environments.

EXAMPLES:

```
sage: from sage.misc.sagedoc import process_dollars
sage: process_dollars('hello')
'hello'
sage: process_dollars('some math: $x=y$')
'some math: `x=y`'
```

Replace \$ with $, and don't do anything when backticks are involved:

```
sage: process_dollars(r'a ``$REAL`` dollar sign: \$')
'a ``$REAL`` dollar sign: $'
```

Don't make any changes on lines indented more than the first nonempty line:

```
sage: s = '\n first line\n     indented $x=y$'
sage: s == process_dollars(s)
True
```

Don't replace dollar signs enclosed in curly braces:

```
sage: process_dollars(r'f(n) = 0 \text{ if $n$ is prime}')
'f(n) = 0 \\text{ if $n$ is prime}'
```

This is not perfect:

```
sage: process_dollars(r'$f(n) = 0 \text{ if $n$ is prime}$')
'`f(n) = 0 \\text{ if $n$ is prime}$'
```

The regular expression search doesn't find the last $. Fortunately, there don't seem to be any instances of this kind of expression in the Sage library, as of this writing.

sage.misc.sagedoc.**process_extlinks**(*s*, *embedded=False*)

In docstrings at the command line, process markup related to the Sphinx extlinks extension. For example, replace `:trac:`NUM`` with `http://trac.sagemath.org/NUM`, and similarly with `:python:TEXT` and `:wikipedia:TEXT`, looking up the url from the dictionary `extlinks` in SAGE_DOC/common/conf.py. If `TEXT` is of the form `blah <LINK>`, then it uses `LINK` rather than `TEXT` to construct the url.

In the notebook, don't do anything: let sphinxify take care of it.

INPUT:

- `s` – string, in practice a docstring
- `embedded` – boolean (optional, default False)

This function is called by `format()`, and if in the notebook, it sets `embedded` to be `True`, otherwise `False`.

EXAMPLES:

```
sage: from sage.misc.sagedoc import process_extlinks
sage: process_extlinks('See :trac:`1234`, :wikipedia:`Wikipedia <Sage_(mathematics_software)>`,
'See http://trac.sagemath.org/1234, http://en.wikipedia.org/wiki/Sage_(mathematics_software), an
sage: process_extlinks('See :trac:`1234` for more information.', embedded=True)
'See :trac:`1234` for more information.'
sage: process_extlinks('see :python:`Implementing Descriptors <reference/datamodel.html#implemen
'see http://docs.python.org/release/.../reference/datamodel.html#implementing-descriptors ...'
```

sage.misc.sagedoc.**process_mathtt**(*s*)

Replace \mathtt{BLAH} with BLAH in the command line.

INPUT:

- `s` - string, in practice a docstring

This function is called by `format()`.

EXAMPLES:

```
sage: from sage.misc.sagedoc import process_mathtt
sage: process_mathtt(r'e^\mathtt{self}')
'e^self'
```

sage.misc.sagedoc.**reference**

The Sage reference manual.

EXAMPLES:

```
sage: reference() # indirect doctest, not tested
sage: manual() # indirect doctest, not tested
```

sage.misc.sagedoc.**search_def**(*name*, *extra1=''*, *extra2=''*, *extra3=''*, *extra4=''*, *extra5=''*, ***kwds*)

Search Sage library source code for function definitions containing `name`. The search is case sensitive.

INPUT: same as for `search_src()`.

OUTPUT: same as for `search_src()`.

Note: The regular expression used by this function only finds function definitions that are preceded by spaces, so if you use tabs on a "def" line, this function will not find it. As tabs are not allowed in Sage library code, this should not be a problem.

EXAMPLES:

See the documentation for search_src() for more examples.

```
sage: print search_def("fetch", interact=False) # random # long time
matrix/matrix0.pyx:    cdef fetch(self, key):
matrix/matrix0.pxd:    cdef fetch(self, key)

sage: print search_def("fetch", path_re="pyx", interact=False) # random # long time
matrix/matrix0.pyx:    cdef fetch(self, key):
```

sage.misc.sagedoc.**search_doc**(*string*, *extra1=''*, *extra2=''*, *extra3=''*, *extra4=''*, *extra5=''*, ***kwds*)

Search Sage HTML documentation for lines containing `string`. The search is case-sensitive.

The file paths in the output are relative to `$SAGE_DOC/output`.

INPUT: same as for search_src().

OUTPUT: same as for search_src().

EXAMPLES:

See the documentation for search_src() for more examples.

```
sage: search_doc('creates a polynomial', path_re='tutorial', interact=False) # random
html/en/tutorial/tour_polynomial.html:<p>This creates a polynomial ring and tells Sage to use (t
```

If you search the documentation for 'tree', then you will get too many results, because many lines in the documentation contain the word 'toctree'. If you use the `whole_word` option, though, you can search for 'tree' without returning all of the instances of 'toctree'. In the following, since `search_doc('tree', interact=False)` returns a string with one line for each match, counting the length of `search_doc('tree', interact=False).splitlines()` gives the number of matches.

```
sage: len(search_doc('tree', interact=False).splitlines()) > 4000  # long time
True
sage: len(search_doc('tree', whole_word=True, interact=False).splitlines()) < 2000  # long time
True
```

sage.misc.sagedoc.**search_src**(*string*, *extra1=''*, *extra2=''*, *extra3=''*, *extra4=''*, *extra5=''*, ***kwds*)

Search Sage library source code for lines containing `string`. The search is case-sensitive.

INPUT:

- `string` - a string to find in the Sage source code.
- `extra1`, ..., `extra5` - additional strings to require when searching. Lines must match all of these, as well as `string`.
- `whole_word` (optional, default False) - if True, search for `string` and `extra1` (etc.) as whole words only. This assumes that each of these arguments is a single word, not a regular expression, and it might have unexpected results if used with regular expressions.
- `ignore_case` (optional, default False) - if True, perform a case-insensitive search
- `multiline` (optional, default False) - if True, search more than one line at a time. In this case, print any matching file names, but don't print line numbers.
- `interact` (optional, default `True`) - if `False`, return a string with all the matches. Otherwise, this function returns `None`, and the results are displayed appropriately, according to whether you are using the notebook or the command-line interface. You should not ordinarily need to use this.
- `path_re` (optional, default '') - regular expression which the filename (including the path) must match.

- module (optional, default 'sage') - the module in which to search. The default is 'sage', the entire Sage library. If module doesn't start with "sage", then the links in the notebook output may not function.

OUTPUT: If interact is False, then return a string with all of the matches, separated by newlines. On the other hand, if interact is True (the default), there is no output. Instead: at the command line, the search results are printed on the screen in the form filename:line_number:line of text, showing the filename in which each match occurs, the line number where it occurs, and the actual matching line. (If multiline is True, then only the filename is printed for each match.) The file paths in the output are relative to $SAGE_SRC. In the notebook, each match produces a link to the actual file in which it occurs.

The string and extraN arguments are treated as regular expressions, as is path_re, and errors will be raised if they are invalid. The matches will be case-sensitive unless ignore_case is True.

Note: The extraN parameters are present only because search_src(string, *extras, interact=False) is not parsed correctly by Python 2.6; see http://bugs.python.org/issue1909.

EXAMPLES:

First note that without using interact=False, this function produces no output, while with interact=False, the output is a string. These examples almost all use this option, so that they have something to which to compare their output.

You can search for "matrix" by typing search_src("matrix"). This particular search will produce many results:

```
sage: len(search_src("matrix", interact=False).splitlines()) # random # long time
9522
```

You can restrict to the Sage calculus code with search_src("matrix", module="sage.calculus"), and this produces many fewer results:

```
sage: len(search_src("matrix", module="sage.calculus", interact=False).splitlines()) # random
26
```

Note that you can do tab completion on the module string. Another way to accomplish a similar search:

```
sage: len(search_src("matrix", path_re="calc", interact=False).splitlines()) > 15
True
```

The following produces an error because the string 'fetch(' is a malformed regular expression:

```
sage: print search_src(" fetch(", "def", interact=False)
Traceback (most recent call last):
...
error: unbalanced parenthesis
```

To fix this, *escape* the parenthesis with a backslash:

```
sage: print search_src(" fetch\(", "def", interact=False) # random # long time
matrix/matrix0.pyx:    cdef fetch(self, key):
matrix/matrix0.pxd:    cdef fetch(self, key)

sage: print search_src(" fetch\(", "def", "pyx", interact=False) # random # long time
matrix/matrix0.pyx:    cdef fetch(self, key):
```

As noted above, the search is case-sensitive, but you can make it case-insensitive with the 'ignore_case' key word:

```
sage: s = search_src('Matrix', path_re='matrix', interact=False); s.find('x') > 0
True
```

```
sage: s = search_src('MatRiX', path_re='matrix', interact=False); s.find('x') > 0
False

sage: s = search_src('MatRiX', path_re='matrix', interact=False, ignore_case=True); s.find('x')
True
```

Searches are by default restricted to single lines, but this can be changed by setting `multiline` to be True. In the following, since `search_src(string, interact=False)` returns a string with one line for each match, counting the length of `search_src(string, interact=False).splitlines()` gives the number of matches.

```
sage: len(search_src('log', 'derivative', interact=False).splitlines()) < 10
True
sage: len(search_src('log', 'derivative', interact=False, multiline=True).splitlines()) > 30
True
```

A little recursive narcissism: let's do a doctest that searches for this function's doctests. Note that you can't put "sage:" in the doctest string because it will get replaced by the Python ">>>" prompt.

```
sage: print search_src('^ *sage[:] .*search_src\(', interact=False) # long time
misc/sagedoc.py:... len(search_src("matrix", interact=False).splitlines()) # random # long time
misc/sagedoc.py:... len(search_src("matrix", module="sage.calculus", interact=False).splitlines(
misc/sagedoc.py:... len(search_src("matrix", path_re="calc", interact=False).splitlines()) > 15
misc/sagedoc.py:... print search_src(" fetch(", "def", interact=False)
misc/sagedoc.py:... print search_src(" fetch\(", "def", interact=False) # random # long time
misc/sagedoc.py:... print search_src(" fetch\(", "def", "pyx", interact=False) # random # long t
misc/sagedoc.py:... s = search_src('Matrix', path_re='matrix', interact=False); s.find('x') > 0
misc/sagedoc.py:... s = search_src('MatRiX', path_re='matrix', interact=False); s.find('x') > 0
misc/sagedoc.py:... s = search_src('MatRiX', path_re='matrix', interact=False, ignore_case=True)
misc/sagedoc.py:... len(search_src('log', 'derivative', interact=False).splitlines()) < 10
misc/sagedoc.py:... len(search_src('log', 'derivative', interact=False, multiline=True).splitlin
misc/sagedoc.py:... print search_src('^ *sage[:] .*search_src\(', interact=False) # long time
misc/sagedoc.py:... len(search_src("matrix", interact=False).splitlines()) > 9000 # long time
misc/sagedoc.py:... print search_src('matrix', 'column', 'row', 'sub', 'start', 'index', interac
```

TESTS:

As of this writing, there are about 9500 lines in the Sage library that contain "matrix"; it seems safe to assume we'll continue to have over 9000 such lines:

```
sage: len(search_src("matrix", interact=False).splitlines()) > 9000 # long time
True
```

Check that you can pass 5 parameters:

```
sage: print search_src('matrix', 'column', 'row', 'sub', 'start', 'index', interact=False) # ran
matrix/matrix0.pyx:598:        Get The 2 x 2 submatrix of M, starting at row index and column
matrix/matrix0.pyx:607:        Get the 2 x 3 submatrix of M starting at row index and column ind
matrix/matrix0.pyx:924:        Set the 2 x 2 submatrix of M, starting at row index and column
matrix/matrix0.pyx:933:        Set the 2 x 3 submatrix of M starting at row index and column
```

`sage.misc.sagedoc.`**`tutorial`**

The Sage tutorial. To get started with Sage, start here.

EXAMPLES:

```
sage: tutorial()   # indirect doctest, not tested
```

CHAPTER

THIRTYFOUR

EDIT THE SOURCE CODE OF SAGE INTERACTIVELY

AUTHORS:

- Nils Bruin
- William Stein – touch up for inclusion in Sage.
- Simon King: Make it usable on extension classes that do not have a docstring; include this module into the reference manual and fix some syntax errors in the doc strings.

This module provides a routine to open the source file of a python object in an editor of your choice, if the source file can be figured out. For files that appear to be from the sage library, the path name gets modified to the corresponding file in the current branch, i.e., the file that gets copied into the library upon 'sage -br'.

The editor to be run, and the way it should be called to open the requested file at the right line number, can be supplied via a template. For a limited number of editors, templates are already known to the system. In those cases it suffices to give the editor name.

In fact, if the environment variable `EDITOR` is set to a known editor, then the system will use that if no template has been set explicitly.

`sage.misc.edit_module.edit`(*obj*, *editor=None*, *bg=None*)

Open source code of obj in editor of your choice.

INPUT:

- editor – str (default: None); If given, use specified editor. Choice is stored for next time.

AUTHOR:

- Nils Bruin (2007-10-03)

EXAMPLES:

This is a typical example of how to use this routine.

```
# make some object obj
sage: edit(obj)    # not tested
```

Now for more details and customization:

```
sage: import sage.misc.edit_module as m
sage: m.set_edit_template("vi -c ${line} ${file}")
```

In fact, since vi is a well-known editor, you could also just use

```
sage: m.set_editor("vi")
```

To illustrate:

```
sage: m.edit_template.template
'vi -c ${line} ${file}'
```

And if your environment variable EDITOR is set to a recognised editor, you would not have to set anything.

To edit the source of an object, just type something like:

```
sage: edit(edit)            # not tested
```

sage.misc.edit_module.**edit_devel**(*self*, *filename*, *linenum*)

This function is for internal use and is called by IPython when you use the IPython commands %edit or %ed.

This hook calls the default implementation, but changes the filename for files that appear to be from the sage library: if the filename begins with 'SAGE_ROOT/local/lib/python.../site-packages', it replaces this by 'SAGE_ROOT/src'.

EXAMPLES:

```
sage: %edit gcd          # indirect doctest, not tested
sage: %ed gcd            # indirect doctest, not tested
```

The above should open your favorite editor (as stored in the environment variable EDITOR) with the file in which gcd is defined, and when your editor supports it, also at the line in wich gcd is defined.

sage.misc.edit_module.**file_and_line**(*obj*)

Look up source file and line number of obj.

If the file lies in the Sage library, the path name of the corresponding file in the current branch (i.e., the file that gets copied into the Sage library upon running 'sage -br'). Note that the first line of a file is considered to be 1 rather than 0 because most editors think that this is the case.

AUTHORS:

- Nils Bruin (2007-10-03)
- Simon King (2011-05): Use sageinspect to get the file and the line.

EXAMPLES:

```
sage: import sage.misc.edit_module as edit_module
sage: edit_module.file_and_line(sage)
('...sage/__init__.py', 0)
```

The following tests against a bug that was fixed in trac ticket #11298:

```
sage: edit_module.file_and_line(x)
('...sage/symbolic/expression.pyx', ...)
```

sage.misc.edit_module.**set_edit_template**(*template_string*)

Sets default edit template string.

It should reference ${file} and ${line}. This routine normally needs to be called prior to using 'edit'. However, if the editor set in the shell variable EDITOR is known, then the system will substitute an appropriate template for you. See edit_module.template_defaults for the recognised templates.

AUTHOR:

- Nils Bruin (2007-10-03)

EXAMPLES:

```
sage: from sage.misc.edit_module import set_edit_template
sage: set_edit_template("echo EDIT ${file}:${line}")
```

```
sage: edit(sage)       # not tested
EDIT /usr/local/sage/src/sage/__init__.py:1
```

`sage.misc.edit_module.set_editor(editor_name, opts='')`

Sets the editor to be used by the edit command by basic editor name.

Currently, the system only knows appropriate call strings for a limited number of editors. If you want to use another editor, you should set the whole edit template via set_edit_template.

AUTHOR:

- Nils Bruin (2007-10-05)

EXAMPLES:

```
sage: from sage.misc.edit_module import set_editor
sage: set_editor('vi')
sage: sage.misc.edit_module.edit_template.template
'vi -c ${line} ${file}'
```

`sage.misc.edit_module.template_fields(template)`

Given a String.Template object, returns the fields.

AUTHOR:

- Nils Bruin (2007-10-22)

EXAMPLES:

```
sage: from sage.misc.edit_module import template_fields
sage: from string import Template
sage: t=Template("Template ${one} with ${two} and ${three}")
sage: template_fields(t)
['three', 'two', 'one']
```

CHAPTER

THIRTYFIVE

MISCELLANEOUS ARITHMETIC FUNCTIONS

`sage.rings.arith.`**`CRT`**`(`*a*, *b*, *m=None*, *n=None*`)`

Returns a solution to a Chinese Remainder Theorem problem.

INPUT:

- `a`, `b` - two residues (elements of some ring for which extended gcd is available), or two lists, one of residues and one of moduli.
- `m`, `n` - (default: `None`) two moduli, or `None`.

OUTPUT:

If `m`, `n` are not `None`, returns a solution x to the simultaneous congruences $x \equiv a \bmod m$ and $x \equiv b \bmod n$, if one exists. By the Chinese Remainder Theorem, a solution to the simultaneous congruences exists if and only if $a \equiv b \pmod{\gcd(m, n)}$. The solution x is only well-defined modulo $\mathrm{lcm}(m, n)$.

If `a` and `b` are lists, returns a simultaneous solution to the congruences $x \equiv a_i \pmod{b_i}$, if one exists.

See also:

- `CRT_list()`

EXAMPLES:

Using `crt` by giving it pairs of residues and moduli:

```
sage: crt(2, 1, 3, 5)
11
sage: crt(13, 20, 100, 301)
28013
sage: crt([2, 1], [3, 5])
11
sage: crt([13, 20], [100, 301])
28013
```

You can also use upper case:

```
sage: c = CRT(2,3, 3, 5); c
8
sage: c % 3 == 2
True
sage: c % 5 == 3
True
```

Note that this also works for polynomial rings:

```
sage: K.<a> = NumberField(x^3 - 7)
sage: R.<y> = K[]
```

```
sage: f = y^2 + 3
sage: g = y^3 - 5
sage: CRT(1,3,f,g)
-3/26*y^4 + 5/26*y^3 + 15/26*y + 53/26
sage: CRT(1,a,f,g)
(-3/52*a + 3/52)*y^4 + (5/52*a - 5/52)*y^3 + (15/52*a - 15/52)*y + 27/52*a + 25/52
```

You can also do this for any number of moduli:

```
sage: K.<a> = NumberField(x^3 - 7)
sage: R.<x> = K[]
sage: CRT([], [])
0
sage: CRT([a], [x])
a
sage: f = x^2 + 3
sage: g = x^3 - 5
sage: h = x^5 + x^2 - 9
sage: k = CRT([1, a, 3], [f, g, h]); k
(127/26988*a - 5807/386828)*x^9 + (45/8996*a - 33677/1160484)*x^8 + (2/173*a - 6/173)*x^7 + (133
sage: k.mod(f)
1
sage: k.mod(g)
a
sage: k.mod(h)
3
```

If the moduli are not coprime, a solution may not exist:

```
sage: crt(4,8,8,12)
20
sage: crt(4,6,8,12)
Traceback (most recent call last):
...
ValueError: No solution to crt problem since gcd(8,12) does not divide 4-6

sage: x = polygen(QQ)
sage: crt(2,3,x-1,x+1)
-1/2*x + 5/2
sage: crt(2,x,x^2-1,x^2+1)
-1/2*x^3 + x^2 + 1/2*x + 1
sage: crt(2,x,x^2-1,x^3-1)
Traceback (most recent call last):
...
ValueError: No solution to crt problem since gcd(x^2 - 1,x^3 - 1) does not divide 2-x

sage: crt(int(2), int(3), int(7), int(11))
58
```

`sage.rings.arith.`**`CRT_basis`**(*moduli*)

Returns a CRT basis for the given moduli.

INPUT:

- **`moduli` - list of pairwise coprime moduli m which admit an** extended Euclidean algorithm

OUTPUT:

- a list of elements a_i of the same length as m such that a_i is congruent to 1 modulo m_i and to 0 modulo m_j for $j \neq i$.

Note: The pairwise coprimality of the input is not checked.

EXAMPLES:

```
sage: a1 = ZZ(mod(42,5))
sage: a2 = ZZ(mod(42,13))
sage: c1,c2 = CRT_basis([5,13])
sage: mod(a1*c1+a2*c2,5*13)
42
```

A polynomial example:

```
sage: x=polygen(QQ)
sage: mods = [x,x^2+1,2*x-3]
sage: b = CRT_basis(mods)
sage: b
[-2/3*x^3 + x^2 - 2/3*x + 1, 6/13*x^3 - x^2 + 6/13*x, 8/39*x^3 + 8/39*x]
sage: [[bi % mj for mj in mods] for bi in b]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

`sage.rings.arith.`**`CRT_list`**`(`*v*, *moduli*`)`

Given a list `v` of elements and a list of corresponding `moduli`, find a single element that reduces to each element of `v` modulo the corresponding moduli.

See also:

- `crt()`

EXAMPLES:

```
sage: CRT_list([2,3,2], [3,5,7])
23
sage: x = polygen(QQ)
sage: c = CRT_list([3], [x]); c
3
sage: c.parent()
Univariate Polynomial Ring in x over Rational Field
```

It also works if the moduli are not coprime:

```
sage: CRT_list([32,2,2],[60,90,150])
452
```

But with non coprime moduli there is not always a solution:

```
sage: CRT_list([32,2,1],[60,90,150])
Traceback (most recent call last):
...
ValueError: No solution to crt problem since gcd(180,150) does not divide 92-1
```

The arguments must be lists:

```
sage: CRT_list([1,2,3],"not a list")
Traceback (most recent call last):
...
ValueError: Arguments to CRT_list should be lists
sage: CRT_list("not a list",[2,3])
Traceback (most recent call last):
...
ValueError: Arguments to CRT_list should be lists
```

The list of moduli must have the same length as the list of elements:

```
sage: CRT_list([1,2,3],[2,3,5])
23
sage: CRT_list([1,2,3],[2,3])
Traceback (most recent call last):
...
ValueError: Arguments to CRT_list should be lists of the same length
sage: CRT_list([1,2,3],[2,3,5,7])
Traceback (most recent call last):
...
ValueError: Arguments to CRT_list should be lists of the same length
```

TESTS:

```
sage: CRT([32r,2r,2r],[60r,90r,150r])
452
```

sage.rings.arith.**CRT_vectors**(*X*, *moduli*)

Vector form of the Chinese Remainder Theorem: given a list of integer vectors v_i and a list of coprime moduli m_i, find a vector w such that $w = v_i \pmod m)_i$ for all i. This is more efficient than applying `CRT()` to each entry.

INPUT:

- `X` - list or tuple, consisting of lists/tuples/vectors/etc of integers of the same length
- `moduli` - list of len(X) moduli

OUTPUT:

- `list` - application of CRT componentwise.

EXAMPLES:

```
sage: CRT_vectors([[3,5,7],[3,5,11]], [2,3])
[3, 5, 5]

sage: CRT_vectors([vector(ZZ, [2,3,1]), Sequence([1,7,8],ZZ)], [8,9])
[10, 43, 17]
```

class sage.rings.arith.**Euler_Phi**

Return the value of the Euler phi function on the integer n. We defined this to be the number of positive integers <= n that are relatively prime to n. Thus if n<=0 then `euler_phi(n)` is defined and equals 0.

INPUT:

- `n` - an integer

EXAMPLES:

```
sage: euler_phi(1)
1
sage: euler_phi(2)
1
sage: euler_phi(3)
2
sage: euler_phi(12)
4
sage: euler_phi(37)
36
```

Notice that euler_phi is defined to be 0 on negative numbers and 0.

```
sage: euler_phi(-1)
0
sage: euler_phi(0)
0
sage: type(euler_phi(0))
<type 'sage.rings.integer.Integer'>
```

We verify directly that the phi function is correct for 21.

```
sage: euler_phi(21)
12
sage: [i for i in range(21) if gcd(21,i) == 1]
[1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20]
```

The length of the list of integers 'i' in range(n) such that the gcd(i,n) == 1 equals euler_phi(n).

```
sage: len([i for i in range(21) if gcd(21,i) == 1]) == euler_phi(21)
True
```

The phi function also has a special plotting method.

```
sage: P = plot(euler_phi, -3, 71)
```

AUTHORS:

- William Stein
- Alex Clemesha (2006-01-10): some examples

plot (*xmin=1*, *xmax=50*, *pointsize=30*, *rgbcolor=(0, 0, 1)*, *join=True*, ***kwds*)
Plot the Euler phi function.

INPUT:

- `xmin` - default: 1
- `xmax` - default: 50
- `pointsize` - default: 30
- `rgbcolor` - default: (0,0,1)
- `join` - default: True; whether to join the points.
- `**kwds` - passed on

EXAMPLES:

```
sage: p = Euler_Phi().plot()
sage: p.ymax()
46.0
```

sage.rings.arith.**GCD** (*a*, *b=None*, ***kwargs*)
The greatest common divisor of a and b, or if a is a list and b is omitted the greatest common divisor of all elements of a.

INPUT:

- `a,b` - two elements of a ring with gcd or
- `a` - a list or tuple of elements of a ring with gcd

Additional keyword arguments are passed to the respectively called methods.

OUTPUT:

The given elements are first coerced into a common parent. Then, their greatest common divisor *in that common parent* is returned.

EXAMPLES:
```
sage: GCD(97,100)
1
sage: GCD(97*10^15, 19^20*97^2)
97
sage: GCD(2/3, 4/5)
2/15
sage: GCD([2,4,6,8])
2
sage: GCD(srange(0,10000,10))  # fast  !!
10
```

Note that to take the gcd of n elements for $n \neq 2$ you must put the elements into a list by enclosing them in `[..]`. Before #4988 the following wrongly returned 3 since the third parameter was just ignored:
```
sage: gcd(3,6,2)
Traceback (most recent call last):
...
TypeError: gcd() takes at most 2 arguments (3 given)
sage: gcd([3,6,2])
1
```

Similarly, giving just one element (which is not a list) gives an error:
```
sage: gcd(3)
Traceback (most recent call last):
...
TypeError: 'sage.rings.integer.Integer' object is not iterable
```

By convention, the gcd of the empty list is (the integer) 0:
```
sage: gcd([])
0
sage: type(gcd([]))
<type 'sage.rings.integer.Integer'>
```

TESTS:

The following shows that indeed coercion takes place before computing the gcd. This behaviour was introduced in trac ticket #10771:
```
sage: R.<x>=QQ[]
sage: S.<x>=ZZ[]
sage: p = S.random_element(degree=(0,10))
sage: q = R.random_element(degree=(0,10))
sage: parent(gcd(1/p,q))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: parent(gcd([1/p,q]))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

Make sure we try QQ and not merely ZZ (trac ticket #13014):
```
sage: bool(gcd(2/5, 3/7) == gcd(SR(2/5), SR(3/7)))
True
```

Make sure that the gcd of Expressions stays symbolic:

```
sage: parent(gcd(2, 4))
Integer Ring
sage: parent(gcd(SR(2), 4))
Symbolic Ring
sage: parent(gcd(2, SR(4)))
Symbolic Ring
sage: parent(gcd(SR(2), SR(4)))
Symbolic Ring
```

Verify that objects without gcd methods but which can't be coerced to ZZ or QQ raise an error:

```
sage: F.<a,b> = FreeMonoid(2)
sage: gcd(a,b)
Traceback (most recent call last):
...
TypeError: unable to find gcd
```

`sage.rings.arith.`**`LCM`**`(`*a*, *b=None*`)`

The least common multiple of a and b, or if a is a list and b is omitted the least common multiple of all elements of a.

Note that LCM is an alias for lcm.

INPUT:

- `a,b` - two elements of a ring with lcm or
- `a` - a list or tuple of elements of a ring with lcm

OUTPUT:

First, the given elements are coerced into a common parent. Then, their least common multiple *in that parent* is returned.

EXAMPLES:

```
sage: lcm(97,100)
9700
sage: LCM(97,100)
9700
sage: LCM(0,2)
0
sage: LCM(-3,-5)
15
sage: LCM([1,2,3,4,5])
60
sage: v = LCM(range(1,10000))    # *very* fast!
sage: len(str(v))
4349
```

TESTS:

The following tests against a bug that was fixed in trac ticket #10771:

```
sage: lcm(4/1,2)
4
```

The following shows that indeed coercion takes place before computing the least common multiple:

```
sage: R.<x>=QQ[]
sage: S.<x>=ZZ[]
sage: p = S.random_element(degree=(0,5))
```

```
sage: q = R.random_element(degree=(0,5))
sage: parent(lcm([1/p,q]))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

Make sure we try QQ and not merely ZZ (trac ticket #13014):

```
sage: bool(lcm(2/5, 3/7) == lcm(SR(2/5), SR(3/7)))
True
```

Make sure that the lcm of Expressions stays symbolic:

```
sage: parent(lcm(2, 4))
Integer Ring
sage: parent(lcm(SR(2), 4))
Symbolic Ring
sage: parent(lcm(2, SR(4)))
Symbolic Ring
sage: parent(lcm(SR(2), SR(4)))
Symbolic Ring
```

Verify that objects without lcm methods but which can't be coerced to ZZ or QQ raise an error:

```
sage: F.<a,b> = FreeMonoid(2)
sage: lcm(a,b)
Traceback (most recent call last):
...
TypeError: unable to find lcm
```

class `sage.rings.arith.`**`Moebius`**

Returns the value of the Moebius function of abs(n), where n is an integer.

DEFINITION: $\mu(n)$ is 0 if n is not square free, and otherwise equals $(-1)^r$, where n has r distinct prime factors.

For simplicity, if $n = 0$ we define $\mu(n) = 0$.

IMPLEMENTATION: Factors or - for integers - uses the PARI C library.

INPUT:

- `n` - anything that can be factored.

OUTPUT: 0, 1, or -1

EXAMPLES:

```
sage: moebius(-5)
-1
sage: moebius(9)
0
sage: moebius(12)
0
sage: moebius(-35)
1
sage: moebius(-1)
1
sage: moebius(7)
-1

sage: moebius(0)    # potentially nonstandard!
0
```

The moebius function even makes sense for non-integer inputs.

```
sage: x = GF(7)['x'].0
sage: moebius(x+2)
-1
```

plot (*xmin=0*, *xmax=50*, *pointsize=30*, *rgbcolor=(0, 0, 1)*, *join=True*, ***kwds*)

Plot the Moebius function.

INPUT:

- `xmin` - default: 0
- `xmax` - default: 50
- `pointsize` - default: 30
- `rgbcolor` - default: (0,0,1)
- `join` - default: True; whether to join the points (very helpful in seeing their order).
- `**kwds` - passed on

EXAMPLES:

```
sage: p = Moebius().plot()
sage: p.ymax()
1.0
```

range (*start*, *stop=None*, *step=None*)

Return the Moebius function evaluated at the given range of values, i.e., the image of the list range(start, stop, step) under the Mobius function.

This is much faster than directly computing all these values with a list comprehension.

EXAMPLES:

```
sage: v = moebius.range(-10,10); v
[1, 0, 0, -1, 1, -1, 0, -1, -1, 1, 0, 1, -1, -1, 0, -1, 1, -1, 0, 0]
sage: v == [moebius(n) for n in range(-10,10)]
True
sage: v = moebius.range(-1000, 2000, 4)
sage: v == [moebius(n) for n in range(-1000,2000, 4)]
True
```

class sage.rings.arith.**Sigma**

Return the sum of the k-th powers of the divisors of n.

INPUT:

- `n` - integer
- `k` - integer (default: 1)

OUTPUT: integer

EXAMPLES:

```
sage: sigma(5)
6
sage: sigma(5,2)
26
```

The sigma function also has a special plotting method.

```
sage: P = plot(sigma, 1, 100)
```

This method also works with k-th powers.

```
sage: P = plot(sigma, 1, 100, k=2)
```

AUTHORS:

- William Stein: original implementation
- Craig Citro (2007-06-01): rewrote for huge speedup

TESTS:

```
sage: sigma(100,4)
106811523
sage: sigma(factorial(100),3).mod(144169)
3672
sage: sigma(factorial(150),12).mod(691)
176
sage: RR(sigma(factorial(133),20))
2.80414775675747e4523
sage: sigma(factorial(100),0)
39001250856960000
sage: sigma(factorial(41),1)
229199532273029988767733858700732906511758707916800
```

plot (*xmin=1*, *xmax=50*, *k=1*, *pointsize=30*, *rgbcolor=(0, 0, 1)*, *join=True*, ***kwds*)

Plot the sigma (sum of k-th powers of divisors) function.

INPUT:

- `xmin` - default: 1
- `xmax` - default: 50
- `k` - default: 1
- `pointsize` - default: 30
- `rgbcolor` - default: (0,0,1)
- `join` - default: True; whether to join the points.
- `**kwds` - passed on

EXAMPLES:

```
sage: p = Sigma().plot()
sage: p.ymax()
124.0
```

`sage.rings.arith.`**XGCD** (*a*, *b*)

Return a triple `(g,s,t)` such that $g = s \cdot a + t \cdot b = \gcd(a, b)$.

Note: One exception is if a and b are not in a principal ideal domain (see Wikipedia article Principal_ideal_domain), e.g., they are both polynomials over the integers. Then this function can't in general return `(g,s,t)` as above, since they need not exist. Instead, over the integers, we first multiply g by a divisor of the resultant of a/g and b/g, up to sign.

INPUT:

- `a, b` - integers or more generally, element of a ring for which the xgcd make sense (e.g. a field or univariate polynomials).

OUTPUT:

- `g, s, t` - such that $g = s \cdot a + t \cdot b$

Note: There is no guarantee that the returned cofactors (s and t) are minimal.

EXAMPLES:

```
sage: xgcd(56, 44)
(4, 4, -5)
sage: 4*56 + (-5)*44
4

sage: g, a, b = xgcd(5/1, 7/1); g, a, b
(1, 3, -2)
sage: a*(5/1) + b*(7/1) == g
True

sage: x = polygen(QQ)
sage: xgcd(x^3 - 1, x^2 - 1)
(x - 1, 1, -x)

sage: K.<g> = NumberField(x^2-3)
sage: g.xgcd(g+2)
(1, 1/3*g, 0)

sage: R.<a,b> = K[]
sage: S.<y> = R.fraction_field()[]
sage: xgcd(y^2, a*y+b)
(1, a^2/b^2, ((-a)/b^2)*y + 1/b)
sage: xgcd((b+g)*y^2, (a+g)*y+b)
(1, (a^2 + (2*g)*a + 3)/(b^3 + (g)*b^2), ((-a + (-g))/b^2)*y + 1/b)
```

Here is an example of a xgcd for two polynomials over the integers, where the linear combination is not the gcd but the gcd multiplied by the resultant:

```
sage: R.<x> = ZZ[]
sage: gcd(2*x*(x-1), x^2)
x
sage: xgcd(2*x*(x-1), x^2)
(2*x, -1, 2)
sage: (2*(x-1)).resultant(x)
2
```

`sage.rings.arith.`**`algdep`**(*z*, *degree*, *known_bits=None*, *use_bits=None*, *known_digits=None*, *use_digits=None*, *height_bound=None*, *proof=False*)

Returns a polynomial of degree at most $degree$ which is approximately satisfied by the number z. Note that the returned polynomial need not be irreducible, and indeed usually won't be if z is a good approximation to an algebraic number of degree less than $degree$.

You can specify the number of known bits or digits of z with `known_bits=k` or `known_digits=k`. PARI is then told to compute the result using $0.8k$ of these bits/digits. Or, you can specify the precision to use directly with `use_bits=k` or `use_digits=k`. If none of these are specified, then the precision is taken from the input value.

A height bound may be specified to indicate the maximum coefficient size of the returned polynomial; if a sufficiently small polynomial is not found, then `None` will be returned. If `proof=True` then the result is returned only if it can be proved correct (i.e. the only possible minimal polynomial satisfying the height bound, or no such polynomial exists). Otherwise a `ValueError` is raised indicating that higher precision is required.

ALGORITHM: Uses LLL for real/complex inputs, PARI C-library `algdep` command otherwise.

Note that `algebraic_dependency` is a synonym for `algdep`.

INPUT:

- `z` - real, complex, or p-adic number
- `degree` - an integer
- **`height_bound` - an integer (default: `None`) specifying the maximum** coefficient size for the returned polynomial
- `proof` - a boolean (default: `False`), requires height_bound to be set

EXAMPLES:

```
sage: algdep(1.888888888888888, 1)
9*x - 17
sage: algdep(0.12121212121212,1)
33*x - 4
sage: algdep(sqrt(2),2)
x^2 - 2
```

This example involves a complex number:

```
sage: z = (1/2)*(1 + RDF(sqrt(3)) *CC.0); z
0.500000000000000 + 0.866025403784439*I
sage: p = algdep(z, 6); p
x^3 + 1
sage: p.factor()
(x + 1) * (x^2 - x + 1)
sage: z^2 - z + 1
0.000000000000000
```

This example involves a p-adic number:

```
sage: K = Qp(3, print_mode = 'series')
sage: a = K(7/19); a
1 + 2*3 + 3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^8 + 2*3^9 + 3^11 + 3^12 + 2*3^15 + 2*3^16 + 3^17 + 2*3^1
sage: algdep(a, 1)
19*x - 7
```

These examples show the importance of proper precision control. We compute a 200-bit approximation to $sqrt(2)$ which is wrong in the 33'rd bit:

```
sage: z = sqrt(RealField(200)(2)) + (1/2)^33
sage: p = algdep(z, 4); p
227004321085*x^4 - 216947902586*x^3 - 99411220986*x^2 + 82234881648*x - 211871195088
sage: factor(p)
227004321085*x^4 - 216947902586*x^3 - 99411220986*x^2 + 82234881648*x - 211871195088
sage: algdep(z, 4, known_bits=32)
x^2 - 2
sage: algdep(z, 4, known_digits=10)
x^2 - 2
sage: algdep(z, 4, use_bits=25)
x^2 - 2
sage: algdep(z, 4, use_digits=8)
x^2 - 2
```

Using the `height_bound` and `proof` parameters, we can see that pi is not the root of an integer polynomial of degree at most 5 and coefficients bounded above by 10:

```
sage: algdep(pi.n(), 5, height_bound=10, proof=True) is None
True
```

For stronger results, we need more precicion:

```
sage: algdep(pi.n(), 5, height_bound=100, proof=True) is None
Traceback (most recent call last):
...
ValueError: insufficient precision for non-existence proof
sage: algdep(pi.n(200), 5, height_bound=100, proof=True) is None
True

sage: algdep(pi.n(), 10, height_bound=10, proof=True) is None
Traceback (most recent call last):
...
ValueError: insufficient precision for non-existence proof
sage: algdep(pi.n(200), 10, height_bound=10, proof=True) is None
True
```

We can also use `proof=True` to get positive results:

```
sage: a = sqrt(2) + sqrt(3) + sqrt(5)
sage: algdep(a.n(), 8, height_bound=1000, proof=True)
Traceback (most recent call last):
...
ValueError: insufficient precision for uniqueness proof
sage: f = algdep(a.n(1000), 8, height_bound=1000, proof=True); f
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
sage: f(a).expand()
0
```

TESTS:

```
sage: algdep(complex("1+2j"), 4)
x^2 - 2*x + 5
```

`sage.rings.arith.`**`algebraic_dependency`**(*z*, *degree*, *known_bits=None*, *use_bits=None*, *known_digits=None*, *use_digits=None*, *height_bound=None*, *proof=False*)

Returns a polynomial of degree at most $degree$ which is approximately satisfied by the number z. Note that the returned polynomial need not be irreducible, and indeed usually won't be if z is a good approximation to an algebraic number of degree less than $degree$.

You can specify the number of known bits or digits of z with `known_bits=k` or `known_digits=k`. PARI is then told to compute the result using $0.8k$ of these bits/digits. Or, you can specify the precision to use directly with `use_bits=k` or `use_digits=k`. If none of these are specified, then the precision is taken from the input value.

A height bound may be specified to indicate the maximum coefficient size of the returned polynomial; if a sufficiently small polynomial is not found, then `None` will be returned. If `proof=True` then the result is returned only if it can be proved correct (i.e. the only possible minimal polynomial satisfying the height bound, or no such polynomial exists). Otherwise a `ValueError` is raised indicating that higher precision is required.

ALGORITHM: Uses LLL for real/complex inputs, PARI C-library `algdep` command otherwise.

Note that `algebraic_dependency` is a synonym for `algdep`.

INPUT:

- `z` - real, complex, or p-adic number
- `degree` - an integer
- **`height_bound` - an integer (default: `None`) specifying the maximum** coefficient size for the returned polynomial

• `proof` - a boolean (default: `False`), requires height_bound to be set

EXAMPLES:

```
sage: algdep(1.888888888888888, 1)
9*x - 17
sage: algdep(0.12121212121212,1)
33*x - 4
sage: algdep(sqrt(2),2)
x^2 - 2
```

This example involves a complex number:

```
sage: z = (1/2)*(1 + RDF(sqrt(3)) *CC.0); z
0.500000000000000 + 0.866025403784439*I
sage: p = algdep(z, 6); p
x^3 + 1
sage: p.factor()
(x + 1) * (x^2 - x + 1)
sage: z^2 - z + 1
0.000000000000000
```

This example involves a p-adic number:

```
sage: K = Qp(3, print_mode = 'series')
sage: a = K(7/19); a
1 + 2*3 + 3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^8 + 2*3^9 + 3^11 + 3^12 + 2*3^15 + 2*3^16 + 3^17 + 2*3^1
sage: algdep(a, 1)
19*x - 7
```

These examples show the importance of proper precision control. We compute a 200-bit approximation to $sqrt(2)$ which is wrong in the 33'rd bit:

```
sage: z = sqrt(RealField(200)(2)) + (1/2)^33
sage: p = algdep(z, 4); p
227004321085*x^4 - 216947902586*x^3 - 99411220986*x^2 + 82234881648*x - 211871195088
sage: factor(p)
227004321085*x^4 - 216947902586*x^3 - 99411220986*x^2 + 82234881648*x - 211871195088
sage: algdep(z, 4, known_bits=32)
x^2 - 2
sage: algdep(z, 4, known_digits=10)
x^2 - 2
sage: algdep(z, 4, use_bits=25)
x^2 - 2
sage: algdep(z, 4, use_digits=8)
x^2 - 2
```

Using the `height_bound` and `proof` parameters, we can see that pi is not the root of an integer polynomial of degree at most 5 and coefficients bounded above by 10:

```
sage: algdep(pi.n(), 5, height_bound=10, proof=True) is None
True
```

For stronger results, we need more precicion:

```
sage: algdep(pi.n(), 5, height_bound=100, proof=True) is None
Traceback (most recent call last):
...
ValueError: insufficient precision for non-existence proof
sage: algdep(pi.n(200), 5, height_bound=100, proof=True) is None
True
```

```
sage: algdep(pi.n(), 10, height_bound=10, proof=True) is None
Traceback (most recent call last):
...
ValueError: insufficient precision for non-existence proof
sage: algdep(pi.n(200), 10, height_bound=10, proof=True) is None
True
```

We can also use `proof=True` to get positive results:

```
sage: a = sqrt(2) + sqrt(3) + sqrt(5)
sage: algdep(a.n(), 8, height_bound=1000, proof=True)
Traceback (most recent call last):
...
ValueError: insufficient precision for uniqueness proof
sage: f = algdep(a.n(1000), 8, height_bound=1000, proof=True); f
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
sage: f(a).expand()
0
```

TESTS:

```
sage: algdep(complex("1+2j"), 4)
x^2 - 2*x + 5
```

`sage.rings.arith.`**`bernoulli`**(*n*, *algorithm='default'*, *num_threads=1*)

Return the n-th Bernoulli number, as a rational number.

INPUT:

- `n` - an integer
- `algorithm`:
  - `'default'` - (default) use 'pari' for n <= 30000, and 'bernmm' for n > 30000 (this is just a heuristic, and not guaranteed to be optimal on all hardware)
  - `'pari'` - use the PARI C library
  - `'gap'` - use GAP
  - `'gp'` - use PARI/GP interpreter
  - `'magma'` - use MAGMA (optional)
  - `'bernmm'` - use bernmm package (a multimodular algorithm)
- `num_threads` - positive integer, number of threads to use (only used for bernmm algorithm)

EXAMPLES:

```
sage: bernoulli(12)
-691/2730
sage: bernoulli(50)
495057205241079648212477525/66
```

We demonstrate each of the alternative algorithms:

```
sage: bernoulli(12, algorithm='gap')
-691/2730
sage: bernoulli(12, algorithm='gp')
-691/2730
sage: bernoulli(12, algorithm='magma')           # optional - magma
-691/2730
sage: bernoulli(12, algorithm='pari')
```

```
-691/2730
sage: bernoulli(12, algorithm='bernmm')
-691/2730
sage: bernoulli(12, algorithm='bernmm', num_threads=4)
-691/2730
```

TESTS:

```
sage: algs = ['gap','gp','pari','bernmm']
sage: test_list = [ZZ.random_element(2, 2255) for _ in range(500)]
sage: vals = [[bernoulli(i,algorithm = j) for j in algs] for i in test_list]  # long time (up to
sage: union([len(union(x))==1 for x in vals])  # long time (depends on previous line)
[True]
sage: algs = ['gp','pari','bernmm']
sage: test_list = [ZZ.random_element(2256, 5000) for _ in range(500)]
sage: vals = [[bernoulli(i,algorithm = j) for j in algs] for i in test_list]  # long time (up to
sage: union([len(union(x))==1 for x in vals])  # long time (depends on previous line)
[True]
```

AUTHOR:

- David Joyner and William Stein

`sage.rings.arith.`**`binomial`**(*x*, *m*, ***kwds*)

Return the binomial coefficient

$$\binom{x}{m} = x(x-1)\cdots(x-m+1)/m!$$

which is defined for $m \in \mathbf{Z}$ and any x. We extend this definition to include cases when $x - m$ is an integer but m is not by

$$\binom{x}{m} = \binom{x}{x-m}$$

If $m < 0$, return 0.

INPUT:

- `x`, `m` - numbers or symbolic expressions. Either `m` or `x-m` must be an integer.

OUTPUT: number or symbolic expression (if input is symbolic)

EXAMPLES:

```
sage: from sage.rings.arith import binomial
sage: binomial(5,2)
10
sage: binomial(2,0)
1
sage: binomial(1/2, 0)
1
sage: binomial(3,-1)
0
sage: binomial(20,10)
184756
sage: binomial(-2, 5)
-6
sage: binomial(-5, -2)
0
sage: binomial(RealField()('2.5'), 2)
```

```
1.87500000000000
sage: n=var('n'); binomial(n,2)
1/2*(n - 1)*n
sage: n=var('n'); binomial(n,n)
1
sage: n=var('n'); binomial(n,n-1)
n
sage: binomial(2^100, 2^100)
1

sage: k, i = var('k,i')
sage: binomial(k,i)
binomial(k, i)
```

If $x \in \mathbf{Z}$, there is an optional 'algorithm' parameter, which can be 'mpir' (faster for small values) or 'pari' (faster for large values):

```
sage: a = binomial(100, 45, algorithm='mpir')
sage: b = binomial(100, 45, algorithm='pari')
sage: a == b
True
```

TESTS:

We test that certain binomials are very fast (this should be instant) – see trac ticket #3309:

```
sage: a = binomial(RR(1140000.78), 23310000)
```

We test conversion of arguments to Integers – see trac ticket #6870:

```
sage: binomial(1/2,1/1)
1/2
sage: binomial(10^20+1/1,10^20)
100000000000000000001
sage: binomial(SR(10**7),10**7)
1
sage: binomial(3/2,SR(1/1))
3/2
```

Some floating point cases – see trac ticket #7562, trac ticket #9633, and trac ticket #12448:

```
sage: binomial(1.,3)
0.000000000000000
sage: binomial(-2.,3)
-4.00000000000000
sage: binomial(0.5r, 5)
0.02734375
sage: a = binomial(float(1001), float(1)); a
1001.0
sage: type(a)
<type 'float'>
sage: binomial(float(1000), 1001)
0.0
```

Test symbolic and uni/multivariate polynomials:

```
sage: K.<x> = ZZ[]
sage: binomial(x,3)
1/6*x^3 - 1/2*x^2 + 1/3*x
sage: binomial(x,3).parent()
```

```
Univariate Polynomial Ring in x over Rational Field
sage: K.<x,y> = Integers(7)[]
sage: binomial(y,3)
-y^3 + 3*y^2 - 2*y
sage: binomial(y,3).parent()
Multivariate Polynomial Ring in x, y over Ring of integers modulo 7
sage: n = var('n')
sage: binomial(n,2)
1/2*(n - 1)*n
```

`sage.rings.arith.binomial_coefficients(n)`

Return a dictionary containing pairs $\{(k_1, k_2) : C_{k,n}\}$ where C_{k_n} are binomial coefficients and $n = k_1 + k_2$.

INPUT:

- `n` - an integer

OUTPUT: dict

EXAMPLES:

```
sage: sorted(binomial_coefficients(3).items())
[((0, 3), 1), ((1, 2), 3), ((2, 1), 3), ((3, 0), 1)]
```

Notice the coefficients above are the same as below:

```
sage: R.<x,y> = QQ[]
sage: (x+y)^3
x^3 + 3*x^2*y + 3*x*y^2 + y^3
```

AUTHORS:

- Fredrik Johansson

`sage.rings.arith.continuant(v, n=None)`

Function returns the continuant of the sequence v (list or tuple).

Definition: see Graham, Knuth and Patashnik, *Concrete Mathematics*, section 6.7: Continuants. The continuant is defined by

- $K_0() = 1$
- $K_1(x_1) = x_1$
- $K_n(x_1, \cdots, x_n) = K_{n-1}(x_n, \cdots x_{n-1})x_n + K_{n-2}(x_1, \cdots, x_{n-2})$

If `n = None` or `n > len(v)` the default `n = len(v)` is used.

INPUT:

- `v` - list or tuple of elements of a ring
- `n` - optional integer

OUTPUT: element of ring (integer, polynomial, etcetera).

EXAMPLES:

```
sage: continuant([1,2,3])
10
sage: p = continuant([2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10])
sage: q = continuant([1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10])
sage: p/q
517656/190435
sage: continued_fraction([2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10]).convergent(14)
```

```
517656/190435
sage: x = PolynomialRing(RationalField(),'x',5).gens()
sage: continuant(x)
x0*x1*x2*x3*x4 + x0*x1*x2 + x0*x1*x4 + x0*x3*x4 + x2*x3*x4 + x0 + x2 + x4
sage: continuant(x, 3)
x0*x1*x2 + x0 + x2
sage: continuant(x,2)
x0*x1 + 1
```

We verify the identity

$$K_n(z, z, \cdots, z) = \sum_{k=0}^{n} \binom{n-k}{k} z^{n-2k}$$

for $n = 6$ using polynomial arithmetic:

```
sage: z = QQ['z'].0
sage: continuant((z,z,z,z,z,z,z,z,z,z,z,z,z,z,z),6)
z^6 + 5*z^4 + 6*z^2 + 1

sage: continuant(9)
Traceback (most recent call last):
...
TypeError: object of type 'sage.rings.integer.Integer' has no len()
```

AUTHORS:

- Jaap Spies (2007-02-06)

`sage.rings.arith.crt(a, b, m=None, n=None)`
Returns a solution to a Chinese Remainder Theorem problem.

INPUT:

- `a`, `b` - two residues (elements of some ring for which extended gcd is available), or two lists, one of residues and one of moduli.
- `m`, `n` - (default: `None`) two moduli, or `None`.

OUTPUT:

If `m`, `n` are not `None`, returns a solution x to the simultaneous congruences $x \equiv a \bmod m$ and $x \equiv b \bmod n$, if one exists. By the Chinese Remainder Theorem, a solution to the simultaneous congruences exists if and only if $a \equiv b \pmod{\gcd(m, n)}$. The solution x is only well-defined modulo $\mathrm{lcm}(m, n)$.

If `a` and `b` are lists, returns a simultaneous solution to the congruences $x \equiv a_i \pmod{b_i}$, if one exists.

See also:

- `CRT_list()`

EXAMPLES:

Using `crt` by giving it pairs of residues and moduli:

```
sage: crt(2, 1, 3, 5)
11
sage: crt(13, 20, 100, 301)
28013
sage: crt([2, 1], [3, 5])
11
```

```
sage: crt([13, 20], [100, 301])
28013
```

You can also use upper case:

```
sage: c = CRT(2,3, 3, 5); c
8
sage: c % 3 == 2
True
sage: c % 5 == 3
True
```

Note that this also works for polynomial rings:

```
sage: K.<a> = NumberField(x^3 - 7)
sage: R.<y> = K[]
sage: f = y^2 + 3
sage: g = y^3 - 5
sage: CRT(1,3,f,g)
-3/26*y^4 + 5/26*y^3 + 15/26*y + 53/26
sage: CRT(1,a,f,g)
(-3/52*a + 3/52)*y^4 + (5/52*a - 5/52)*y^3 + (15/52*a - 15/52)*y + 27/52*a + 25/52
```

You can also do this for any number of moduli:

```
sage: K.<a> = NumberField(x^3 - 7)
sage: R.<x> = K[]
sage: CRT([], [])
0
sage: CRT([a], [x])
a
sage: f = x^2 + 3
sage: g = x^3 - 5
sage: h = x^5 + x^2 - 9
sage: k = CRT([1, a, 3], [f, g, h]); k
(127/26988*a - 5807/386828)*x^9 + (45/8996*a - 33677/1160484)*x^8 + (2/173*a - 6/173)*x^7 + (133
sage: k.mod(f)
1
sage: k.mod(g)
a
sage: k.mod(h)
3
```

If the moduli are not coprime, a solution may not exist:

```
sage: crt(4,8,8,12)
20
sage: crt(4,6,8,12)
Traceback (most recent call last):
...
ValueError: No solution to crt problem since gcd(8,12) does not divide 4-6

sage: x = polygen(QQ)
sage: crt(2,3,x-1,x+1)
-1/2*x + 5/2
sage: crt(2,x,x^2-1,x^2+1)
-1/2*x^3 + x^2 + 1/2*x + 1
sage: crt(2,x,x^2-1,x^3-1)
Traceback (most recent call last):
...
```

```
ValueError: No solution to crt problem since gcd(x^2 - 1,x^3 - 1) does not divide 2-x

sage: crt(int(2), int(3), int(7), int(11))
58
```

sage.rings.arith.**dedekind_sum**(*p*, *q*, *algorithm='default'*)

Return the Dedekind sum $s(p,q)$ defined for integers p, q as

$$s(p,q) = \sum_{i=0}^{q-1} \left(\left(\frac{i}{q}\right)\right)\left(\left(\frac{pi}{q}\right)\right)$$

where

$$((x)) = \begin{cases} x - \lfloor x \rfloor - \frac{1}{2} & \text{if } x \in \mathbf{Q} \setminus \mathbf{Z} \\ 0 & \text{if } x \in \mathbf{Z}. \end{cases}$$

Warning: Caution is required as the Dedekind sum sometimes depends on the algorithm or is left undefined when p and q are not coprime.

INPUT:

- `p, q` – integers
- `algorithm` – must be one of the following
  - `'default'` - (default) use FLINT
  - `'flint'` - use FLINT
  - `'pari'` - use PARI (gives different results if p and q are not coprime)

OUTPUT: a rational number

EXAMPLES:

Several small values:

```
sage: for q in range(10): print [dedekind_sum(p,q) for p in range(q+1)]
[0]
[0, 0]
[0, 0, 0]
[0, 1/18, -1/18, 0]
[0, 1/8, 0, -1/8, 0]
[0, 1/5, 0, 0, -1/5, 0]
[0, 5/18, 1/18, 0, -1/18, -5/18, 0]
[0, 5/14, 1/14, -1/14, 1/14, -1/14, -5/14, 0]
[0, 7/16, 1/8, 1/16, 0, -1/16, -1/8, -7/16, 0]
[0, 14/27, 4/27, 1/18, -4/27, 4/27, -1/18, -4/27, -14/27, 0]
```

Check relations for restricted arguments:

```
sage: q = 23; dedekind_sum(1, q); (q-1)*(q-2)/(12*q)
77/46
77/46
sage: p, q = 100, 723    # must be coprime
sage: dedekind_sum(p, q) + dedekind_sum(q, p)
31583/86760
sage: -1/4 + (p/q + q/p + 1/(p*q))/12
31583/86760
```

We check that evaluation works with large input:

```
sage: dedekind_sum(3^54 - 1, 2^93 + 1)
459340694971839990630374299870/29710560942849126597578981379
sage: dedekind_sum(3^54 - 1, 2^93 + 1, algorithm='pari')
459340694971839990630374299870/29710560942849126597578981379
```

We check consistency of the results:

```
sage: dedekind_sum(5, 7, algorithm='default')
-1/14
sage: dedekind_sum(5, 7, algorithm='flint')
-1/14
sage: dedekind_sum(5, 7, algorithm='pari')
-1/14
sage: dedekind_sum(6, 8, algorithm='default')
-1/8
sage: dedekind_sum(6, 8, algorithm='flint')
-1/8
sage: dedekind_sum(6, 8, algorithm='pari')
-1/8
```

REFERENCES:

- Wikipedia article Dedekind_sum

`sage.rings.arith.`**`differences`**(*lis*, *n=1*)

Returns the n successive differences of the elements in lis.

EXAMPLES:

```
sage: differences(prime_range(50))
[1, 2, 2, 4, 2, 4, 2, 4, 6, 2, 6, 4, 2, 4]
sage: differences([i^2 for i in range(1,11)])
[3, 5, 7, 9, 11, 13, 15, 17, 19]
sage: differences([i^3 + 3*i for i in range(1,21)])
[10, 22, 40, 64, 94, 130, 172, 220, 274, 334, 400, 472, 550, 634, 724, 820, 922, 1030, 1144]
sage: differences([i^3 - i^2 for i in range(1,21)], 2)
[10, 16, 22, 28, 34, 40, 46, 52, 58, 64, 70, 76, 82, 88, 94, 100, 106, 112]
sage: differences([p - i^2 for i, p in enumerate(prime_range(50))], 3)
[-1, 2, -4, 4, -4, 4, 0, -6, 8, -6, 0, 4]
```

AUTHORS:

- Timothy Clemans (2008-03-09)

`sage.rings.arith.`**`divisors`**(*n*)

Returns a list of all positive integer divisors of the nonzero integer n.

INPUT:

- `n` - the element

EXAMPLES:

```
sage: divisors(-3)
[1, 3]
sage: divisors(6)
[1, 2, 3, 6]
sage: divisors(28)
[1, 2, 4, 7, 14, 28]
sage: divisors(2^5)
[1, 2, 4, 8, 16, 32]
```

```
sage: divisors(100)
[1, 2, 4, 5, 10, 20, 25, 50, 100]
sage: divisors(1)
[1]
sage: divisors(0)
Traceback (most recent call last):
...
ValueError: n must be nonzero
sage: divisors(2^3 * 3^2 * 17)
[1, 2, 3, 4, 6, 8, 9, 12, 17, 18, 24, 34, 36, 51, 68, 72, 102, 136, 153, 204, 306, 408, 612, 122
```

This function works whenever one has unique factorization:

```
sage: K.<a> = QuadraticField(7)
sage: divisors(K.ideal(7))
[Fractional ideal (1), Fractional ideal (a), Fractional ideal (7)]
sage: divisors(K.ideal(3))
[Fractional ideal (1), Fractional ideal (3), Fractional ideal (-a + 2), Fractional ideal (-a - 2
sage: divisors(K.ideal(35))
[Fractional ideal (1), Fractional ideal (5), Fractional ideal (a), Fractional ideal (7), Fractic
```

TESTS:

```
sage: divisors(int(300))
[1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 25, 30, 50, 60, 75, 100, 150, 300]
```

`sage.rings.arith.`**`eratosthenes`**`(`*n*`)`

Return a list of the primes $\leq n$.

This is extremely slow and is for educational purposes only.

INPUT:

- `n` - a positive integer

OUTPUT:

- a list of primes less than or equal to n.

EXAMPLES:

```
sage: len(eratosthenes(100))
25
sage: eratosthenes(3)
[2, 3]
```

`sage.rings.arith.`**`factor`**`(`*n*, *proof=None*, *int_=False*, *algorithm='pari'*, *verbose=0*, ***kwds*`)`

Returns the factorization of `n`. The result depends on the type of `n`.

If `n` is an integer, returns the factorization as an object of type `Factorization`.

If `n` is not an integer, `n.factor(proof=proof, **kwds)` gets called. See `n.factor??` for more documentation in this case.

Warning: This means that applying `factor` to an integer result of a symbolic computation will not factor the integer, because it is considered as an element of a larger symbolic ring.

EXAMPLE:

```
sage: f(n)=n^2
sage: is_prime(f(3))
False
sage: factor(f(3))
9
```

INPUT:

- `n` - an nonzero integer
- `proof` - bool or None (default: None)
- `int_` - bool (default: False) whether to return answers as Python ints
- `algorithm` - string
  - `'pari'` - (default) use the PARI c library
  - `'kash'` - use KASH computer algebra system (requires the optional kash package be installed)
  - `'magma'` - use Magma (requires magma be installed)
- `verbose` - integer (default: 0); PARI's debug variable is set to this; e.g., set to 4 or 8 to see lots of output during factorization.

OUTPUT:

- factorization of n

The qsieve and ecm commands give access to highly optimized implementations of algorithms for doing certain integer factorization problems. These implementations are not used by the generic factor command, which currently just calls PARI (note that PARI also implements sieve and ecm algorithms, but they aren't as optimized). Thus you might consider using them instead for certain numbers.

The factorization returned is an element of the class `Factorization`; see Factorization?? for more details, and examples below for usage. A Factorization contains both the unit factor (+1 or -1) and a sorted list of (prime, exponent) pairs.

The factorization displays in pretty-print format but it is easy to obtain access to the (prime,exponent) pairs and the unit, to recover the number from its factorization, and even to multiply two factorizations. See examples below.

EXAMPLES:

```
sage: factor(500)
2^2 * 5^3
sage: factor(-20)
-1 * 2^2 * 5
sage: f=factor(-20)
sage: list(f)
[(2, 2), (5, 1)]
sage: f.unit()
-1
sage: f.value()
-20
sage: factor( -next_prime(10^2) * next_prime(10^7) )
-1 * 101 * 10000019
```

```
sage: factor(-500, algorithm='kash')      # optional - kash
-1 * 2^2 * 5^3

sage: factor(-500, algorithm='magma')     # optional - magma
-1 * 2^2 * 5^3

sage: factor(0)
Traceback (most recent call last):
...
ArithmeticError: Prime factorization of 0 not defined.
sage: factor(1)
1
sage: factor(-1)
-1
sage: factor(2^(2^7)+1)
59649589127497217 * 5704689200685129054721
```

Sage calls PARI's factor, which has proof False by default. Sage has a global proof flag, set to True by default (see `sage.structure.proof.proof`, or proof.[tab]). To override the default, call this function with proof=False.

```
sage: factor(3^89-1, proof=False)
2 * 179 * 1611479891519807 * 5042939439565996049162197

sage: factor(2^197 + 1)  # long time (2s)
3 * 197002597249 * 1348959352853811313 * 251951573867253012259144010843
```

Any object which has a factor method can be factored like this:

```
sage: K.<i> = QuadraticField(-1)
sage: factor(122 - 454*i)
(-3*i - 2) * (-i - 2)^3 * (i + 1)^3 * (i + 4)
```

To access the data in a factorization:

```
sage: f = factor(420); f
2^2 * 3 * 5 * 7
sage: [x for x in f]
[(2, 2), (3, 1), (5, 1), (7, 1)]
sage: [p for p,e in f]
[2, 3, 5, 7]
sage: [e for p,e in f]
[2, 1, 1, 1]
sage: [p^e for p,e in f]
[4, 3, 5, 7]
```

`sage.rings.arith.`**`factorial`**(*n*, *algorithm='gmp'*)

Compute the factorial of n, which is the product $1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$.

INPUT:

- `n` - an integer
- `algorithm` - string (default: 'gmp'):
  - `'gmp'` - use the GMP C-library factorial function
  - `'pari'` - use PARI's factorial function

OUTPUT: an integer

EXAMPLES:

```
sage: from sage.rings.arith import factorial
sage: factorial(0)
1
sage: factorial(4)
24
sage: factorial(10)
3628800
sage: factorial(1) == factorial(0)
True
sage: factorial(6) == 6*5*4*3*2
True
sage: factorial(1) == factorial(0)
True
sage: factorial(71) == 71* factorial(70)
True
sage: factorial(-32)
Traceback (most recent call last):
...
ValueError: factorial -- must be nonnegative
```

PERFORMANCE: This discussion is valid as of April 2006. All timings below are on a Pentium Core Duo 2Ghz MacBook Pro running Linux with a 2.6.16.1 kernel.

- It takes less than a minute to compute the factorial of 10^7 using the GMP algorithm, and the factorial of 10^6 takes less than 4 seconds.
- The GMP algorithm is faster and more memory efficient than the PARI algorithm. E.g., PARI computes 10^7 factorial in 100 seconds on the core duo 2Ghz.
- For comparison, computation in Magma $\leq$ 2.12-10 of $n!$ is best done using `*[1..n]`. It takes 113 seconds to compute the factorial of 10^7 and 6 seconds to compute the factorial of 10^6. Mathematica V5.2 compute the factorial of 10^7 in 136 seconds and the factorial of 10^6 in 7 seconds. (Mathematica is notably very efficient at memory usage when doing factorial calculations.)

`sage.rings.arith.`**`falling_factorial`**`(x, a)`

Returns the falling factorial $(x)_a$.

The notation in the literature is a mess: often $(x)_a$, but there are many other notations: GKP: Concrete Mathematics uses $x^{\underline{a}}$.

Definition: for integer $a \geq 0$ we have $x(x-1)\cdots(x-a+1)$. In all other cases we use the GAMMA-function: $\frac{\Gamma(x+1)}{\Gamma(x-a+1)}$.

INPUT:

- `x` - element of a ring
- `a` - a non-negative integer or

OR

- `x and a` - any numbers

OUTPUT: the falling factorial

EXAMPLES:

```
sage: falling_factorial(10, 3)
720
sage: falling_factorial(10, RR('3.0'))
720.000000000000
sage: falling_factorial(10, RR('3.3'))
```

```
1310.11633396601
sage: falling_factorial(10, 10)
3628800
sage: factorial(10)
3628800
sage: a = falling_factorial(1+I, I); a
gamma(I + 2)
sage: CC(a)
0.652965496420167 + 0.343065839816545*I
sage: falling_factorial(1+I, 4)
4*I + 2
sage: falling_factorial(I, 4)
-10

sage: M = MatrixSpace(ZZ, 4, 4)
sage: A = M([1,0,1,0,1,0,1,0,1,0,10,10,1,0,1,1])
sage: falling_factorial(A, 2) # A(A - I)
[  1   0  10  10]
[  1   0  10  10]
[ 20   0 101 100]
[  2   0  11  10]

sage: x = ZZ['x'].0
sage: falling_factorial(x, 4)
x^4 - 6*x^3 + 11*x^2 - 6*x
```

TESTS:

Check that trac ticket #14858 is fixed:

```
sage: falling_factorial(-4, SR(2))
20
```

Check that trac ticket #16770 is fixed:

```
sage: d = var('d')
sage: type(falling_factorial(d, 0))
<type 'sage.symbolic.expression.Expression'>
```

AUTHORS:

- Jaap Spies (2006-03-05)

`sage.rings.arith.`**`four_squares`**`(n)`

Write the integer n as a sum of four integer squares.

INPUT:

- n – an integer

OUTPUT: a tuple (a, b, c, d) of non-negative integers such that $n = a^2 + b^2 + c^2 + d^2$ with $a <= b <= c <= d$.

EXAMPLES:

```
sage: four_squares(3)
(0, 1, 1, 1)
sage: four_squares(13)
(0, 0, 2, 3)
sage: four_squares(130)
(0, 0, 3, 11)
sage: four_squares(1101011011004)
(90, 102, 1220, 1049290)
```

```
sage: four_squares(10^100-1)
(155024616290, 2612183768627, 14142135623730950488016887, 999999999999999999999999999999999999
sage: for i in range(2^129, 2^129+10000):  # long time
....:     S = four_squares(i)
....:     assert sum(x^2 for x in S) == i
```

TESTS:

```
sage: for _ in xrange(100):
....:     n = ZZ.random_element(2**32,2**34)
....:     aa,bb,cc,dd = four_squares(n)
....:     assert aa**2 + bb**2 + cc**2 + dd**2 == n
```

`sage.rings.arith.`**`fundamental_discriminant`**`(`*D*`)`

Return the discriminant of the quadratic extension $K = Q(\sqrt{D})$, i.e. an integer d congruent to either 0 or 1, mod 4, and such that, at most, the only square dividing it is 4.

INPUT:

- `D` - an integer

OUTPUT:

- an integer, the fundamental discriminant

EXAMPLES:

```
sage: fundamental_discriminant(102)
408
sage: fundamental_discriminant(720)
5
sage: fundamental_discriminant(2)
8
```

`sage.rings.arith.`**`gcd`**`(`*a*, *b=None*, ***kwargs*`)`

The greatest common divisor of a and b, or if a is a list and b is omitted the greatest common divisor of all elements of a.

INPUT:

- `a,b` - two elements of a ring with gcd or
- `a` - a list or tuple of elements of a ring with gcd

Additional keyword arguments are passed to the respectively called methods.

OUTPUT:

The given elements are first coerced into a common parent. Then, their greatest common divisor *in that common parent* is returned.

EXAMPLES:

```
sage: GCD(97,100)
1
sage: GCD(97*10^15, 19^20*97^2)
97
sage: GCD(2/3, 4/5)
2/15
sage: GCD([2,4,6,8])
2
sage: GCD(srange(0,10000,10))  # fast  !!
10
```

Note that to take the gcd of n elements for $n \neq 2$ you must put the elements into a list by enclosing them in `[..]`. Before #4988 the following wrongly returned 3 since the third parameter was just ignored:

```
sage: gcd(3,6,2)
Traceback (most recent call last):
...
TypeError: gcd() takes at most 2 arguments (3 given)
sage: gcd([3,6,2])
1
```

Similarly, giving just one element (which is not a list) gives an error:

```
sage: gcd(3)
Traceback (most recent call last):
...
TypeError: 'sage.rings.integer.Integer' object is not iterable
```

By convention, the gcd of the empty list is (the integer) 0:

```
sage: gcd([])
0
sage: type(gcd([]))
<type 'sage.rings.integer.Integer'>
```

TESTS:

The following shows that indeed coercion takes place before computing the gcd. This behaviour was introduced in trac ticket #10771:

```
sage: R.<x>=QQ[]
sage: S.<x>=ZZ[]
sage: p = S.random_element(degree=(0,10))
sage: q = R.random_element(degree=(0,10))
sage: parent(gcd(1/p,q))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: parent(gcd([1/p,q]))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

Make sure we try QQ and not merely ZZ (trac ticket #13014):

```
sage: bool(gcd(2/5, 3/7) == gcd(SR(2/5), SR(3/7)))
True
```

Make sure that the gcd of Expressions stays symbolic:

```
sage: parent(gcd(2, 4))
Integer Ring
sage: parent(gcd(SR(2), 4))
Symbolic Ring
sage: parent(gcd(2, SR(4)))
Symbolic Ring
sage: parent(gcd(SR(2), SR(4)))
Symbolic Ring
```

Verify that objects without gcd methods but which can't be coerced to ZZ or QQ raise an error:

```
sage: F.<a,b> = FreeMonoid(2)
sage: gcd(a,b)
Traceback (most recent call last):
...
TypeError: unable to find gcd
```

sage.rings.arith.**get_gcd**(*order*)

Return the fastest gcd function for integers of size no larger than order.

EXAMPLES:

```
sage: sage.rings.arith.get_gcd(4000)
<built-in method gcd_int of sage.rings.fast_arith.arith_int object at ...>
sage: sage.rings.arith.get_gcd(400000)
<built-in method gcd_longlong of sage.rings.fast_arith.arith_llong object at ...>
sage: sage.rings.arith.get_gcd(4000000000)
<function gcd at ...>
```

sage.rings.arith.**get_inverse_mod**(*order*)

Return the fastest inverse_mod function for integers of size no larger than order.

EXAMPLES:

```
sage: sage.rings.arith.get_inverse_mod(6000)
<built-in method inverse_mod_int of sage.rings.fast_arith.arith_int object at ...>
sage: sage.rings.arith.get_inverse_mod(600000)
<built-in method inverse_mod_longlong of sage.rings.fast_arith.arith_llong object at ...>
sage: sage.rings.arith.get_inverse_mod(6000000000)
<function inverse_mod at ...>
```

sage.rings.arith.**hilbert_conductor**(*a*, *b*)

This is the product of all (finite) primes where the Hilbert symbol is -1. What is the same, this is the (reduced) discriminant of the quaternion algebra (a, b) over $\mathbf{Q}$.

INPUT:

- a, b – integers

OUTPUT:

- squarefree positive integer

EXAMPLES:

```
sage: hilbert_conductor(-1, -1)
2
sage: hilbert_conductor(-1, -11)
11
sage: hilbert_conductor(-2, -5)
5
sage: hilbert_conductor(-3, -17)
17
```

AUTHOR:

- Gonzalo Tornaria (2009-03-02)

sage.rings.arith.**hilbert_conductor_inverse**(*d*)

Finds a pair of integers (a, b) such that `hilbert_conductor(a,b) == d`. The quaternion algebra (a, b) over $\mathbf{Q}$ will then have (reduced) discriminant d.

INPUT:

- d – square-free positive integer

OUTPUT: pair of integers

EXAMPLES:

```
sage: hilbert_conductor_inverse(2)
(-1, -1)
sage: hilbert_conductor_inverse(3)
(-1, -3)
sage: hilbert_conductor_inverse(6)
(-1, 3)
sage: hilbert_conductor_inverse(30)
(-3, -10)
sage: hilbert_conductor_inverse(4)
Traceback (most recent call last):
...
ValueError: d needs to be squarefree
sage: hilbert_conductor_inverse(-1)
Traceback (most recent call last):
...
ValueError: d needs to be positive
```

AUTHOR:

- Gonzalo Tornaria (2009-03-02)

TESTS:

```
sage: for i in xrange(100):
...     d = ZZ.random_element(2**32).squarefree_part()
...     if hilbert_conductor(*hilbert_conductor_inverse(d)) != d:
...         print "hilbert_conductor_inverse failed for d =", d
```

`sage.rings.arith.`**`hilbert_symbol`**(*a*, *b*, *p*, *algorithm='pari'*)

Returns 1 if $ax^2 + by^2$ p-adically represents a nonzero square, otherwise returns -1. If either a or b is 0, returns 0.

INPUT:

- `a, b` - integers
- `p` - integer; either prime or -1 (which represents the archimedean place)
- `algorithm` - string
  - `'pari'` - (default) use the PARI C library
  - `'direct'` - use a Python implementation
  - `'all'` - use both PARI and direct and check that the results agree, then return the common answer

OUTPUT: integer (0, -1, or 1)

EXAMPLES:

```
sage: hilbert_symbol (-1, -1, -1, algorithm='all')
-1
sage: hilbert_symbol (2,3, 5, algorithm='all')
1
sage: hilbert_symbol (4, 3, 5, algorithm='all')
1
sage: hilbert_symbol (0, 3, 5, algorithm='all')
0
sage: hilbert_symbol (-1, -1, 2, algorithm='all')
-1
sage: hilbert_symbol (1, -1, 2, algorithm='all')
1
sage: hilbert_symbol (3, -1, 2, algorithm='all')
```

```
-1

sage: hilbert_symbol(QQ(-1)/QQ(4), -1, 2) == -1
True
sage: hilbert_symbol(QQ(-1)/QQ(4), -1, 3) == 1
True
```

AUTHORS:

- William Stein and David Kohel (2006-01-05)

sage.rings.arith.**integer_ceil**(*x*)

Return the ceiling of x.

EXAMPLES:

```
sage: integer_ceil(5.4)
6
```

sage.rings.arith.**integer_floor**(*x*)

Return the largest integer $\leq x$.

INPUT:

- x - an object that has a floor method or is coercible to int

OUTPUT: an Integer

EXAMPLES:

```
sage: integer_floor(5.4)
5
sage: integer_floor(float(5.4))
5
sage: integer_floor(-5/2)
-3
sage: integer_floor(RDF(-5/2))
-3
```

sage.rings.arith.**inverse_mod**(*a*, *m*)

The inverse of the ring element a modulo m.

If no special inverse_mod is defined for the elements, it tries to coerce them into integers and perform the inversion there

```
sage: inverse_mod(7,1)
0
sage: inverse_mod(5,14)
3
sage: inverse_mod(3,-5)
2
```

sage.rings.arith.**is_power_of_two**(*n*)

This function returns True if and only if n is a power of 2

INPUT:

- n - integer

OUTPUT:

- True - if n is a power of 2
- False - if not

EXAMPLES:

```
sage: is_power_of_two(1024)
True

sage: is_power_of_two(1)
True

sage: is_power_of_two(24)
False

sage: is_power_of_two(0)
False

sage: is_power_of_two(-4)
False
```

AUTHORS:

- Jaap Spies (2006-12-09)

`sage.rings.arith.is_prime(n)`

Return `True` if n is a prime number, and `False` otherwise.

AUTHORS:

- Kevin Stueve kstueve@uw.edu (2010-01-17): delegated calculation to `n.is_prime()`

INPUT:

- `n` - the object for which to determine primality

EXAMPLES:

```
sage: is_prime(389)
True
sage: is_prime(2000)
False
sage: is_prime(2)
True
sage: is_prime(-1)
False
sage: is_prime(1)
False
sage: is_prime(-2)
False
```

`sage.rings.arith.is_prime_power(n, flag=None, get_data=False)`

Test whether `n` is a positive power of a prime number

INPUT:

- `n` – an integer
- `get_data` – if set to `True`, return a pair `(p,k)` such that this integer equals `p^k` instead of `True` or `(self,0)` instead of `False`

EXAMPLES:

```
sage: is_prime_power(389)
True
sage: is_prime_power(2000)
False
sage: is_prime_power(2)
```

```
True
sage: is_prime_power(1024)
True
sage: is_prime_power(1024, get_data=True)
(2, 10)

sage: is_prime_power(-1)
False
sage: is_prime_power(1)
False
sage: is_prime_power(QQ(997^100))
True
sage: is_prime_power(1/2197)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
sage: is_prime_power("foo")
Traceback (most recent call last):
...
TypeError: unable to convert 'foo' to an integer
```

sage.rings.arith.**is_pseudoprime**(*n*, *flag=None*)

Test whether n is a pseudo-prime

The result is *NOT* proven correct - *this is a pseudo-primality test!*.

INPUT:

- n – an integer

Note: We do not consider negatives of prime numbers as prime.

EXAMPLES:

```
sage: is_pseudoprime(389)
True
sage: is_pseudoprime(2000)
False
sage: is_pseudoprime(2)
True
sage: is_pseudoprime(-1)
False
sage: factor(-6)
-1 * 2 * 3
sage: is_pseudoprime(1)
False
sage: is_pseudoprime(-2)
False
```

TESTS:

Deprecation warning from trac ticket #16878:

```
sage: is_pseudoprime(127, flag=0)
doctest:...: DeprecationWarning: the keyword 'flag' is deprecated and no longer used
See http://trac.sagemath.org/16878 for details.
True
```

sage.rings.arith.**is_pseudoprime_power**(*n*, *get_data=False*)

Test if n is a power of a pseudoprime.

The result is *NOT* proven correct - *this IS a pseudo-primality test!*. Note that a prime power is a positive power of a prime number so that 1 is not a prime power.

INPUT:

- `n` - an integer
- `get_data` - (boolean) instead of a boolean return a pair (p, k) so that `n` equals p^k and p is a pseudoprime or $(n, 0)$ otherwise.

EXAMPLES:

```
sage: is_pseudoprime_power(389)
True
sage: is_pseudoprime_power(2000)
False
sage: is_pseudoprime_power(2)
True
sage: is_pseudoprime_power(1024)
True
sage: is_pseudoprime_power(-1)
False
sage: is_pseudoprime_power(1)
False
sage: is_pseudoprime_power(997^100)
True
```

Use of the get_data keyword:

```
sage: is_pseudoprime_power(3^1024, get_data=True)
(3, 1024)
sage: is_pseudoprime_power(2^256, get_data=True)
(2, 256)
sage: is_pseudoprime_power(31, get_data=True)
(31, 1)
sage: is_pseudoprime_power(15, get_data=True)
(15, 0)
```

`sage.rings.arith.`**`is_pseudoprime_small_power`**(*n*, *bound=None*, *get_data=False*)

Deprecated version of `is_pseudoprime_power`.

EXAMPLES:

```
sage: is_pseudoprime_small_power(1234)
doctest:...: DeprecationWarning: the function is_pseudoprime_small_power() is deprecated, use is
See http://trac.sagemath.org/16878 for details.
False
sage: is_pseudoprime_small_power(3^1024, get_data=True)
[(3, 1024)]
```

`sage.rings.arith.`**`is_square`**(*n*, *root=False*)

Returns whether or not n is square, and if n is a square also returns the square root. If n is not square, also returns None.

INPUT:

- `n` - an integer
- `root` - whether or not to also return a square root (default: False)

OUTPUT:

- `bool` - whether or not a square

- `object` - (optional) an actual square if found, and None otherwise.

EXAMPLES:

```
sage: is_square(2)
False
sage: is_square(4)
True
sage: is_square(2.2)
True
sage: is_square(-2.2)
False
sage: is_square(CDF(-2.2))
True
sage: is_square((x-1)^2)
True

sage: is_square(4, True)
(True, 2)
```

`sage.rings.arith.`**`is_squarefree`**`(`*n*`)`

Returns True if and only if n is not divisible by the square of an integer > 1.

EXAMPLES:

```
sage: is_squarefree(100)
False
sage: is_squarefree(101)
True
```

`sage.rings.arith.`**`jacobi_symbol`**`(`*a*, *b*`)`

The Jacobi symbol of integers a and b, where b is odd.

Note: The `kronecker_symbol()` command extends the Jacobi symbol to all integers b.

If

$b = p_1^{e_1} * ... * p_r^{e_r}$

then

$(a|b) = (a|p_1)^{e_1} ... (a|p_r)^{e_r}$

where $(a|p_j)$ are Legendre Symbols.

INPUT:

- `a` - an integer
- `b` - an odd integer

EXAMPLES:

```
sage: jacobi_symbol(10,777)
-1
sage: jacobi_symbol(10,5)
0
sage: jacobi_symbol(10,2)
Traceback (most recent call last):
...
ValueError: second input must be odd, 2 is not odd
```

`sage.rings.arith.`**`kronecker`**`(x, y)`

Synonym for `kronecker_symbol()`.

The Kronecker symbol $(x|y)$.

INPUT:

- `x` - integer
- `y` - integer

OUTPUT:

- an integer

EXAMPLES:

```
sage: kronecker(3,5)
-1
sage: kronecker(3,15)
0
sage: kronecker(2,15)
1
sage: kronecker(-2,15)
-1
sage: kronecker(2/3,5)
1
```

`sage.rings.arith.`**`kronecker_symbol`**`(x, y)`

The Kronecker symbol $(x|y)$.

INPUT:

- `x` - integer
- `y` - integer

EXAMPLES:

```
sage: kronecker_symbol(13,21)
-1
sage: kronecker_symbol(101,4)
1
```

IMPLEMENTATION: Using GMP.

`sage.rings.arith.`**`lcm`**`(a, b=None)`

The least common multiple of a and b, or if a is a list and b is omitted the least common multiple of all elements of a.

Note that LCM is an alias for lcm.

INPUT:

- `a,b` - two elements of a ring with lcm or
- `a` - a list or tuple of elements of a ring with lcm

OUTPUT:

First, the given elements are coerced into a common parent. Then, their least common multiple *in that parent* is returned.

EXAMPLES:

```
sage: lcm(97,100)
9700
sage: LCM(97,100)
9700
sage: LCM(0,2)
0
sage: LCM(-3,-5)
15
sage: LCM([1,2,3,4,5])
60
sage: v = LCM(range(1,10000))    # *very* fast!
sage: len(str(v))
4349
```

TESTS:

The following tests against a bug that was fixed in trac ticket #10771:

```
sage: lcm(4/1,2)
4
```

The following shows that indeed coercion takes place before computing the least common multiple:

```
sage: R.<x>=QQ[]
sage: S.<x>=ZZ[]
sage: p = S.random_element(degree=(0,5))
sage: q = R.random_element(degree=(0,5))
sage: parent(lcm([1/p,q]))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

Make sure we try QQ and not merely ZZ (trac ticket #13014):

```
sage: bool(lcm(2/5, 3/7) == lcm(SR(2/5), SR(3/7)))
True
```

Make sure that the lcm of Expressions stays symbolic:

```
sage: parent(lcm(2, 4))
Integer Ring
sage: parent(lcm(SR(2), 4))
Symbolic Ring
sage: parent(lcm(2, SR(4)))
Symbolic Ring
sage: parent(lcm(SR(2), SR(4)))
Symbolic Ring
```

Verify that objects without lcm methods but which can't be coerced to ZZ or QQ raise an error:

```
sage: F.<a,b> = FreeMonoid(2)
sage: lcm(a,b)
Traceback (most recent call last):
...
TypeError: unable to find lcm
```

`sage.rings.arith.`**`legendre_symbol`**`(x, p)`

The Legendre symbol $(x|p)$, for p prime.

Note: The `kronecker_symbol()` command extends the Legendre symbol to composite moduli and $p = 2$.

INPUT:

- x - integer
- p - an odd prime number

EXAMPLES:

```
sage: legendre_symbol(2,3)
-1
sage: legendre_symbol(1,3)
1
sage: legendre_symbol(1,2)
Traceback (most recent call last):
...
ValueError: p must be odd
sage: legendre_symbol(2,15)
Traceback (most recent call last):
...
ValueError: p must be a prime
sage: kronecker_symbol(2,15)
1
sage: legendre_symbol(2/3,7)
-1
```

sage.rings.arith.**mqrr_rational_reconstruction**(*u*, *m*, *T*)

Maximal Quotient Rational Reconstruction.

For research purposes only - this is pure Python, so slow.

INPUT:

- u, m, T - integers such that $m > u \ge 0, T > 0$.

OUTPUT:

Either integers n, d such that $d > 0, \gcd(n, d) = 1, n/d = u \bmod m$, and $T \cdot d \cdot |n| < m$, or None.

Reference: Monagan, Maximal Quotient Rational Reconstruction: An Almost Optimal Algorithm for Rational Reconstruction (page 11)

This algorithm is probabilistic.

EXAMPLES:

```
sage: mqrr_rational_reconstruction(21,3100,13)
(21, 1)
```

sage.rings.arith.**multinomial**(**ks*)

Return the multinomial coefficient

INPUT:

- An arbitrary number of integer arguments $k_1, \ldots, k_n$
- A list of integers $[k_1, \ldots, k_n]$

OUTPUT:

Returns the integer:

$$\binom{k_1 + \cdots + k_n}{k_1, \cdots, k_n} = \frac{(\sum_{i=1}^n k_i)!}{\prod_{i=1}^n k_i!} = \prod_{i=1}^n \binom{\sum_{j=1}^i k_j}{k_i}$$

EXAMPLES:

```
sage: multinomial(0, 0, 2, 1, 0, 0)
3
sage: multinomial([0, 0, 2, 1, 0, 0])
3
sage: multinomial(3, 2)
10
sage: multinomial(2^30, 2, 1)
618970023101454657175683075
sage: multinomial([2^30, 2, 1])
618970023101454657175683075
```

AUTHORS:

- Gabriel Ebner

sage.rings.arith.**multinomial_coefficients**(*m*, *n*)

Return a dictionary containing pairs $\{(k_1, k_2, ..., k_m) : C_{k,n}\}$ where $C_{k,n}$ are multinomial coefficients such that $n = k_1 + k_2 + ... + k_m$.

INPUT:

- `m` - integer
- `n` - integer

OUTPUT: dict

EXAMPLES:

```
sage: sorted(multinomial_coefficients(2, 5).items())
[((0, 5), 1), ((1, 4), 5), ((2, 3), 10), ((3, 2), 10), ((4, 1), 5), ((5, 0), 1)]
```

Notice that these are the coefficients of $(x + y)^5$:

```
sage: R.<x,y> = QQ[]
sage: (x+y)^5
x^5 + 5*x^4*y + 10*x^3*y^2 + 10*x^2*y^3 + 5*x*y^4 + y^5

sage: sorted(multinomial_coefficients(3, 2).items())
[((0, 0, 2), 1), ((0, 1, 1), 2), ((0, 2, 0), 1), ((1, 0, 1), 2), ((1, 1, 0), 2), ((2, 0, 0), 1)]
```

ALGORITHM: The algorithm we implement for computing the multinomial coefficients is based on the following result:

..math:

```
\binom{n}{k_1, \cdots, k_m} =
\frac{k_1+1}{n-k_1}\sum_{i=2}^m \binom{n}{k_1+1, \cdots, k_i-1, \cdots}
```

e.g.:

```
sage: k = (2, 4, 1, 0, 2, 6, 0, 0, 3, 5, 7, 1) # random value
sage: n = sum(k)
sage: s = 0
sage: for i in range(1, len(k)):
...       ki = list(k)
...       ki[0] += 1
...       ki[i] -= 1
...       s += multinomial(n, *ki)
sage: multinomial(n, *k) == (k[0] + 1) / (n - k[0]) * s
True
```

TESTS:

```
sage: multinomial_coefficients(0, 0)
{(): 1}
sage: multinomial_coefficients(0, 3)
{}
```

sage.rings.arith.**next_prime**(*n*, *proof=None*)

The next prime greater than the integer n. If n is prime, then this function does not return n, but the next prime after n. If the optional argument proof is False, this function only returns a pseudo-prime, as defined by the PARI nextprime function. If it is None, uses the global default (see `sage.structure.proof.proof`)

INPUT:

- `n` - integer
- `proof` - bool or None (default: None)

EXAMPLES:

```
sage: next_prime(-100)
2
sage: next_prime(1)
2
sage: next_prime(2)
3
sage: next_prime(3)
5
sage: next_prime(4)
5
```

Notice that the next_prime(5) is not 5 but 7.

```
sage: next_prime(5)
7
sage: next_prime(2004)
2011
```

sage.rings.arith.**next_prime_power**(*n*)

The next prime power greater than the integer n. If n is a prime power, then this function does not return n, but the next prime power after n.

EXAMPLES:

```
sage: next_prime_power(-10)
2
sage: next_prime_power(0)
2
sage: next_prime_power(1)
2
sage: next_prime_power(2)
3
sage: next_prime_power(10)
11
sage: next_prime_power(7)
8
sage: next_prime_power(99)
101
```

sage.rings.arith.**next_probable_prime**(*n*)

Returns the next probable prime after self, as determined by PARI.

INPUT:

- n - an integer

EXAMPLES:

```
sage: next_probable_prime(-100)
2
sage: next_probable_prime(19)
23
sage: next_probable_prime(int(999999999))
1000000007
sage: next_probable_prime(2^768)
155251809230070893514897948846250255525688601711669661113905203802605095268637688633087840882864
```

sage.rings.arith.**nth_prime**(*n*)

Return the n-th prime number (1-indexed, so that 2 is the 1st prime.)

INPUT:

- n – a positive integer

OUTPUT:

- the n-th prime number

EXAMPLES:

```
sage: nth_prime(3)
5
sage: nth_prime(10)
29
```

```
sage: nth_prime(0)
Traceback (most recent call last):
...
ValueError: nth prime meaningless for non-positive n (=0)
```

TESTS:

```
sage: all(prime_pi(nth_prime(j)) == j for j in range(1, 1000, 10))
True
```

sage.rings.arith.**number_of_divisors**(*n*)

Return the number of divisors of the integer n.

INPUT:

- n - a nonzero integer

OUTPUT:

- an integer, the number of divisors of n

EXAMPLES:

```
sage: number_of_divisors(100)
9
sage: number_of_divisors(-720)
30
```

sage.rings.arith.**odd_part**(*n*)

The odd part of the integer n. This is $n/2^v$, where $v = \mathrm{valuation}(n, 2)$.

EXAMPLES:

```
sage: odd_part(5)
5
sage: odd_part(4)
1
sage: odd_part(factorial(31))
122529844256906551386796875
```

sage.rings.arith.**power_mod**(*a*, *n*, *m*)

The n-th power of a modulo the integer m.

EXAMPLES:

```
sage: power_mod(0,0,5)
Traceback (most recent call last):
...
ArithmeticError: 0^0 is undefined.
sage: power_mod(2,390,391)
285
sage: power_mod(2,-1,7)
4
sage: power_mod(11,1,7)
4
sage: R.<x> = ZZ[]
sage: power_mod(3*x, 10, 7)
4*x^10

sage: power_mod(11,1,0)
Traceback (most recent call last):
...
ZeroDivisionError: modulus must be nonzero.
```

sage.rings.arith.**previous_prime**(*n*)

The largest prime < n. The result is provably correct. If n <= 1, this function raises a ValueError.

EXAMPLES:

```
sage: previous_prime(10)
7
sage: previous_prime(7)
5
sage: previous_prime(8)
7
sage: previous_prime(7)
5
sage: previous_prime(5)
3
sage: previous_prime(3)
2
sage: previous_prime(2)
Traceback (most recent call last):
...
ValueError: no previous prime
sage: previous_prime(1)
Traceback (most recent call last):
...
ValueError: no previous prime
sage: previous_prime(-20)
Traceback (most recent call last):
...
ValueError: no previous prime
```

sage.rings.arith.**previous_prime_power**(*n*)

The largest prime power $< n$. The result is provably correct. If $n \leq 2$, this function returns $-x$, where x is prime power and $-x < n$ and no larger negative of a prime power has this property.

EXAMPLES:

```
sage: previous_prime_power(3)
2
sage: previous_prime_power(10)
9
sage: previous_prime_power(7)
5
sage: previous_prime_power(127)
125
```

```
sage: previous_prime_power(2)
Traceback (most recent call last):
...
ValueError: no previous prime power
sage: previous_prime_power(-10)
Traceback (most recent call last):
...
ValueError: no previous prime power
```

```
sage: n = previous_prime_power(2^16 - 1)
sage: while is_prime(n):
....:  n = previous_prime_power(n)
sage: factor(n)
251^2
```

sage.rings.arith.**prime_divisors**(*n*)

The prime divisors of `n`.

INPUT:

- `n` – any object which can be factored

OUTPUT:

A list of prime factors of `n`. For integers, this list is sorted in increasing order.

EXAMPLES:

```
sage: prime_divisors(1)
[]
sage: prime_divisors(100)
[2, 5]
sage: prime_divisors(2004)
[2, 3, 167]
```

If `n` is negative, we do *not* include -1 among the prime divisors, since -1 is not a prime number:

```
sage: prime_divisors(-100)
[2, 5]
```

For polynomials we get all irreducible factors:

```
sage: R.<x> = PolynomialRing(QQ)
sage: prime_divisors(x^12 - 1)
[x - 1, x + 1, x^2 - x + 1, x^2 + 1, x^2 + x + 1, x^4 - x^2 + 1]
```

`sage.rings.arith.`**`prime_factors`**(*n*)

The prime divisors of `n`.

INPUT:

•`n` – any object which can be factored

OUTPUT:

A list of prime factors of `n`. For integers, this list is sorted in increasing order.

EXAMPLES:

```
sage: prime_divisors(1)
[]
sage: prime_divisors(100)
[2, 5]
sage: prime_divisors(2004)
[2, 3, 167]
```

If `n` is negative, we do *not* include -1 among the prime divisors, since -1 is not a prime number:

```
sage: prime_divisors(-100)
[2, 5]
```

For polynomials we get all irreducible factors:

```
sage: R.<x> = PolynomialRing(QQ)
sage: prime_divisors(x^12 - 1)
[x - 1, x + 1, x^2 - x + 1, x^2 + 1, x^2 + x + 1, x^4 - x^2 + 1]
```

`sage.rings.arith.`**`prime_powers`**(*start*, *stop=None*)

List of all positive primes powers between `start` and `stop`-1, inclusive. If the second argument is omitted, returns the prime powers up to the first argument.

INPUT:

•`start` - an integer. If two inputs are given, a lower bound for the returned set of prime powers. If this is the only input, then it is an upper bound.

•`stop` - an integer (default: `None`) An upper bound for the returned set of prime powers.

OUTPUT:

The set of all prime powers between `start` and `stop` or, if only one argument is passed, the set of all prime powers between 1 and `start`. Note that we will here say that the number n is a prime power if $n = p^k$, where p is a prime number and k is a nonnegative integer. Thus, 1 is a prime power, as $1 = 2^0$.

EXAMPLES:

```
sage: prime_powers(20)
[1, 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17, 19]
sage: len(prime_powers(1000))
194
sage: len(prime_range(1000))
168
sage: a = [z for z in range(95,1234) if is_prime_power(z)]
sage: b = prime_powers(95,1234)
sage: len(b)
194
sage: len(a)
194
sage: a[:10]
[97, 101, 103, 107, 109, 113, 121, 125, 127, 128]
```

```
sage: b[:10]
[97, 101, 103, 107, 109, 113, 121, 125, 127, 128]
sage: a == b
True
sage: prime_powers(10,7)
[]
sage: prime_powers(-5)
[]
sage: prime_powers(-1,2)
[1]
```

TESTS:

```
sage: v = prime_powers(10)
sage: type(v[0])      # trac #922
<type 'sage.rings.integer.Integer'>

sage: prime_powers(0,1)
[]
sage: prime_powers(0,2)
[1]

sage: prime_powers("foo")
Traceback (most recent call last):
...
TypeError: start must be an integer, foo is not an integer

sage: prime_powers(6, "bar")
Traceback (most recent call last):
...
TypeError: stop must be an integer, bar is not an integer
```

sage.rings.arith.**prime_to_m_part**(*n*, *m*)

Returns the prime-to-m part of n, i.e., the largest divisor of n that is coprime to m.

INPUT:

- `n` - Integer (nonzero)
- `m` - Integer

OUTPUT: Integer

EXAMPLES:

```
sage: z = 43434
sage: z.prime_to_m_part(20)
21717
```

sage.rings.arith.**primes**(*start*, *stop=None*, *proof=None*)

Returns an iterator over all primes between start and stop-1, inclusive. This is much slower than `prime_range`, but potentially uses less memory. As with `next_prime()`, the optional argument proof controls whether the numbers returned are guaranteed to be prime or not.

This command is like the xrange command, except it only iterates over primes. In some cases it is better to use primes than `prime_range`, because primes does not build a list of all primes in the range in memory all at once. However, it is potentially much slower since it simply calls the `next_prime()` function repeatedly, and `next_prime()` is slow.

INPUT:

- `start` - an integer - lower bound for the primes
- `stop` - an integer (or infinity) optional argument - giving upper (open) bound for the primes
- `proof` - bool or None (default: None) If True, the function yields only proven primes. If False, the function uses a pseudo-primality test, which is much faster for really big numbers but does not provide a proof of primality. If None, uses the global default (see `sage.structure.proof.proof`)

OUTPUT:

- an iterator over primes from start to stop-1, inclusive

EXAMPLES:

```
sage: for p in primes(5,10):
...     print p
...
5
7
sage: list(primes(13))
[2, 3, 5, 7, 11]
sage: list(primes(10000000000, 10000000100))
[10000000019, 10000000033, 10000000061, 10000000069, 10000000097]
sage: max(primes(10^100, 10^100+10^4, proof=False))
1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
sage: next(p for p in primes(10^20, infinity) if is_prime(2*p+1))
100000000000000001243
```

TESTS:

```
sage: for a in range(-10, 50):
...    for b in range(-10, 50):
...        assert list(primes(a,b)) == list(filter(is_prime, xrange(a,b)))
...
sage: sum(primes(-10, 9973, proof=False)) == sum(filter(is_prime, range(-10, 9973)))
True
sage: for p in primes(10, infinity):
...    if p > 20: break
...    print p
...
11
13
17
19
sage: next(p for p in primes(10,oo)) # checks alternate infinity notation
11
```

sage.rings.arith.**primes_first_n**(*n*, *leave_pari=False*)

Return the first n primes.

INPUT:

- n - a nonnegative integer

OUTPUT:

- a list of the first n prime numbers.

EXAMPLES:

```
sage: primes_first_n(10)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
sage: len(primes_first_n(1000))
1000
```

```
sage: primes_first_n(0)
[]
```

sage.rings.arith.**primitive_root**(*n*, *check=True*)

Return a positive integer that generates the multiplicative group of integers modulo n, if one exists; otherwise, raise a `ValueError`.

A primitive root exists if $n = 4$ or $n = p^k$ or $n = 2p^k$, where p is an odd prime and k is a nonnegative number.

INPUT:

- `n` – a non-zero integer
- `check` – bool (default: True); if False, then n is assumed to be a positive integer possessing a primitive root, and behavior is undefined otherwise.

OUTPUT:

A primitive root of n. If n is prime, this is the smallest primitive root.

EXAMPLES:

```
sage: primitive_root(23)
5
sage: primitive_root(-46)
5
sage: primitive_root(25)
2
sage: print [primitive_root(p) for p in primes(100)]
[1, 2, 2, 3, 2, 2, 3, 2, 5, 2, 3, 2, 6, 3, 5, 2, 2, 2, 2, 7, 5, 3, 2, 3, 5]
sage: primitive_root(8)
Traceback (most recent call last):
...
ValueError: no primitive root
```

Note: It takes extra work to check if n has a primitive root; to avoid this, use `check=False`, which may slightly speed things up (but could also result in undefined behavior). For example, the second call below is an order of magnitude faster than the first:

```
sage: n = 10^50 + 151    # a prime
sage: primitive_root(n)
11
sage: primitive_root(n, check=False)
11
```

TESTS:

Various special cases:

```
sage: primitive_root(-1)
0
sage: primitive_root(0)
Traceback (most recent call last):
...
ValueError: no primitive root
sage: primitive_root(1)
0
sage: primitive_root(2)
1
sage: primitive_root(3)
```

```
2
sage: primitive_root(4)
3
```

We test that various numbers without primitive roots give an error - see Trac 10836:

```
sage: primitive_root(15)
Traceback (most recent call last):
...
ValueError: no primitive root
sage: primitive_root(16)
Traceback (most recent call last):
...
ValueError: no primitive root
sage: primitive_root(1729)
Traceback (most recent call last):
...
ValueError: no primitive root
sage: primitive_root(4*7^8)
Traceback (most recent call last):
...
ValueError: no primitive root
```

sage.rings.arith.**quadratic_residues**(*n*)

Return a sorted list of all squares modulo the integer n in the range $0 \leq x < |n|$.

EXAMPLES:

```
sage: quadratic_residues(11)
[0, 1, 3, 4, 5, 9]
sage: quadratic_residues(1)
[0]
sage: quadratic_residues(2)
[0, 1]
sage: quadratic_residues(8)
[0, 1, 4]
sage: quadratic_residues(-10)
[0, 1, 4, 5, 6, 9]
sage: v = quadratic_residues(1000); len(v);
159
```

sage.rings.arith.**radical**(*n*, **args*, ***kwds*)

Return the product of the prime divisors of n.

This calls `n.radical(*args, **kwds)`. If that doesn't work, it does `n.factor(*args, **kwds)` and returns the product of the prime factors in the resulting factorization.

EXAMPLES:

```
sage: radical(2 * 3^2 * 5^5)
30
sage: radical(0)
Traceback (most recent call last):
...
ArithmeticError: Radical of 0 not defined.
sage: K.<i> = QuadraticField(-1)
sage: radical(K(2))
i + 1
```

The next example shows how to compute the radical of a number, assuming no prime > 100000 has exponent >

1 in the factorization:

```
sage: n = 2^1000-1; n / radical(n, limit=100000)
125
```

`sage.rings.arith.`**`random_prime`**(*n*, *proof=None*, *lbound=2*)

Returns a random prime p between $lbound$ and n (i.e. $lbound <= p <= n$). The returned prime is chosen uniformly at random from the set of prime numbers less than or equal to n.

INPUT:

- `n` - an integer >= 2.
- `proof` - bool or None (default: None) If False, the function uses a pseudo-primality test, which is much faster for really big numbers but does not provide a proof of primality. If None, uses the global default (see `sage.structure.proof.proof`)
- `lbound` - an integer >= 2 lower bound for the chosen primes

EXAMPLES:

```
sage: random_prime(100000)
88237
sage: random_prime(2)
2
```

Here we generate a random prime between 100 and 200:

```
sage: random_prime(200, lbound=100)
149
```

If all we care about is finding a pseudo prime, then we can pass in `proof=False`

```
sage: random_prime(200, proof=False, lbound=100)
149
```

TESTS:

```
sage: type(random_prime(2))
<type 'sage.rings.integer.Integer'>
sage: type(random_prime(100))
<type 'sage.rings.integer.Integer'>
sage: random_prime(1, lbound=-2)   #caused Sage hang #10112
Traceback (most recent call last):
...
ValueError: n must be greater than or equal to 2
sage: random_prime(126, lbound=114)
Traceback (most recent call last):
...
ValueError: There are no primes between 114 and 126 (inclusive)
```

AUTHORS:

- Jon Hanke (2006-08-08): with standard Stein cleanup
- Jonathan Bober (2007-03-17)

`sage.rings.arith.`**`rational_reconstruction`**(*a*, *m*, *algorithm='fast'*)

This function tries to compute x/y, where x/y is a rational number in lowest terms such that the reduction of x/y modulo m is equal to a and the absolute values of x and y are both $\leq \sqrt{m/2}$. If such x/y exists, that pair is unique and this function returns it. If no such pair exists, this function raises ZeroDivisionError.

An efficient algorithm for computing rational reconstruction is very similar to the extended Euclidean algorithm. For more details, see Knuth, Vol 2, 3rd ed, pages 656-657.

INPUT:

- `a` – an integer
- `m` – a modulus
- `algorithm` – (default: 'fast')
  - `'fast'` - a fast implementation using direct MPIR calls in Cython.

OUTPUT:

Numerator and denominator n, d of the unique rational number $r = n/d$, if it exists, with n and $|d| \leq \sqrt{N/2}$. Return $(0, 0)$ if no such number exists.

The algorithm for rational reconstruction is described (with a complete nontrivial proof) on pages 656-657 of Knuth, Vol 2, 3rd ed. as the solution to exercise 51 on page 379. See in particular the conclusion paragraph right in the middle of page 657, which describes the algorithm thus:

> This discussion proves that the problem can be solved efficiently by applying Algorithm 4.5.2X with $u = m$ and $v = a$, but with the following replacement for step X2: If $v3 \leq \sqrt{m/2}$, the algorithm terminates. The pair $(x, y) = (|v2|, v3 * \mathrm{sign}(v2))$ is then the unique solution, provided that x and y are coprime and $x \leq \sqrt{m/2}$; otherwise there is no solution. (Alg 4.5.2X is the extended Euclidean algorithm.)

Knuth remarks that this algorithm is due to Wang, Kornerup, and Gregory from around 1983.

EXAMPLES:

```
sage: m = 100000
sage: (119*inverse_mod(53,m))%m
11323
sage: rational_reconstruction(11323,m)
119/53

sage: rational_reconstruction(400,1000)
Traceback (most recent call last):
...
ArithmeticError: rational reconstruction of 400 (mod 1000) does not exist

sage: rational_reconstruction(3, 292393)
3
sage: a = Integers(292393)(45/97); a
204977
sage: rational_reconstruction(a, 292393, algorithm='fast')
45/97
sage: rational_reconstruction(293048, 292393)
Traceback (most recent call last):
...
ArithmeticError: rational reconstruction of 655 (mod 292393) does not exist
sage: rational_reconstruction(0, 0)
Traceback (most recent call last):
...
ZeroDivisionError: rational reconstruction with zero modulus
sage: rational_reconstruction(0, 1, algorithm="foobar")
Traceback (most recent call last):
...
ValueError: unknown algorithm 'foobar'
```

`sage.rings.arith.`**`rising_factorial`**`(x, a)`

Returns the rising factorial $(x)^a$.

The notation in the literature is a mess: often $(x)^a$, but there are many other notations: GKP: Concrete Mathematics uses $x^{\overline{a}}$.

The rising factorial is also known as the Pochhammer symbol, see Maple and Mathematica.

Definition: for integer $a \geq 0$ we have $x(x+1)\cdots(x+a-1)$. In all other cases we use the GAMMA-function: $\frac{\Gamma(x+a)}{\Gamma(x)}$.

INPUT:

- x - element of a ring
- a - a non-negative integer or
- x and a - any numbers

OUTPUT: the rising factorial

EXAMPLES:

```
sage: rising_factorial(10,3)
1320

sage: rising_factorial(10,RR('3.0'))
1320.00000000000

sage: rising_factorial(10,RR('3.3'))
2826.38895824964

sage: a = rising_factorial(1+I, I); a
gamma(2*I + 1)/gamma(I + 1)
sage: CC(a)
0.266816390637832 + 0.122783354006372*I

sage: a = rising_factorial(I, 4); a
-10
```

See falling_factorial(I, 4).

```
sage: x = polygen(ZZ)
sage: rising_factorial(x, 4)
x^4 + 6*x^3 + 11*x^2 + 6*x
```

TESTS:

Check that trac ticket #14858 is fixed:

```
sage: bool(rising_factorial(-4, 2) ==
....:      rising_factorial(-4, SR(2)) ==
....:      rising_factorial(SR(-4), SR(2)))
True
```

Check that trac ticket #16770 is fixed:

```
sage: d = var('d')
sage: type(rising_factorial(d, 0))
<type 'sage.symbolic.expression.Expression'>
```

AUTHORS:

- Jaap Spies (2006-03-05)

sage.rings.arith.**sort_complex_numbers_for_display**(*nums*)

Given a list of complex numbers (or a list of tuples, where the first element of each tuple is a complex number), we sort the list in a "pretty" order. First come the real numbers (with zero imaginary part), then the complex numbers sorted according to their real part. If two complex numbers have a real part which is sufficiently close, then they are sorted according to their imaginary part.

This is not a useful function mathematically (not least because there's no principled way to determine whether the real components should be treated as equal or not). It is called by various polynomial root-finders; its purpose is to make doctest printing more reproducible.

We deliberately choose a cumbersome name for this function to discourage use, since it is mathematically meaningless.

EXAMPLES:

```
sage: import sage.rings.arith
sage: sort_c = sort_complex_numbers_for_display
sage: nums = [CDF(i) for i in range(3)]
sage: for i in range(3):
....:     nums.append(CDF(i + RDF.random_element(-3e-11, 3e-11),
....:                     RDF.random_element()))
....:     nums.append(CDF(i + RDF.random_element(-3e-11, 3e-11),
....:                     RDF.random_element()))
sage: shuffle(nums)
sage: sort_c(nums)
[0.0, 1.0, 2.0, -2.862406201002009e-11 - 0.7088740263015161*I, 2.2108362706985576e-11 - 0.436810
```

sage.rings.arith.**squarefree_divisors**(*x*)

Iterator over the squarefree divisors (up to units) of the element x.

Depends on the output of the prime_divisors function.

INPUT:

```
x -- an element of any ring for which the prime_divisors
     function works.
```

EXAMPLES:

```
sage: list(squarefree_divisors(7))
[1, 7]
sage: list(squarefree_divisors(6))
[1, 2, 3, 6]
sage: list(squarefree_divisors(12))
[1, 2, 3, 6]
```

sage.rings.arith.**subfactorial**(*n*)

Subfactorial or rencontres numbers, or derangements: number of permutations of n elements with no fixed points.

INPUT:

- `n` - non negative integer

OUTPUT:

- `integer` - function value

EXAMPLES:

```
sage: subfactorial(0)
1
sage: subfactorial(1)
0
sage: subfactorial(8)
14833
```

AUTHORS:

- Jaap Spies (2007-01-23)

`sage.rings.arith.`**`sum_of_k_squares`**(*k*, *n*)

Write the integer n as a sum of k integer squares if possible; otherwise raise a `ValueError`.

INPUT:

- `k` – a non-negative integer
- `n` – an integer

OUTPUT: a tuple $(x_1, ..., x_k)$ of non-negative integers such that their squares sum to n.

EXAMPLES:

```
sage: sum_of_k_squares(2, 9634)
(15, 97)
sage: sum_of_k_squares(3, 9634)
(0, 15, 97)
sage: sum_of_k_squares(4, 9634)
(1, 2, 5, 98)
sage: sum_of_k_squares(5, 9634)
(0, 1, 2, 5, 98)
sage: sum_of_k_squares(6, 11^1111-1)
(19215400822645944253860920437586326284, 37204645194585992174252915693267578306, 347365481947739
sage: sum_of_k_squares(7, 0)
(0, 0, 0, 0, 0, 0, 0)
sage: sum_of_k_squares(30,999999)
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 3, 7, 44, 999)
sage: sum_of_k_squares(1, 9)
(3,)
sage: sum_of_k_squares(1, 10)
Traceback (most recent call last):
...
ValueError: 10 is not a sum of 1 square
sage: sum_of_k_squares(1, -10)
Traceback (most recent call last):
...
ValueError: -10 is not a sum of 1 square
sage: sum_of_k_squares(0, 9)
Traceback (most recent call last):
...
ValueError: 9 is not a sum of 0 squares
sage: sum_of_k_squares(0, 0)
()
sage: sum_of_k_squares(7, -1)
Traceback (most recent call last):
...
ValueError: -1 is not a sum of 7 squares
sage: sum_of_k_squares(-1, 0)
Traceback (most recent call last):
...
ValueError: k = -1 must be non-negative
```

`sage.rings.arith.`**`three_squares`**(*n*)

Write the integer n as a sum of three integer squares if possible; otherwise raise a `ValueError`.

INPUT:

- `n` – an integer

OUTPUT: a tuple (a, b, c) of non-negative integers such that $n = a^2 + b^2 + c^2$ with $a <= b <= c$.

EXAMPLES:

```
sage: three_squares(389)
(1, 8, 18)
sage: three_squares(946)
(9, 9, 28)
sage: three_squares(2986)
(3, 24, 49)
sage: three_squares(7^100)
(0, 0, 1798465042647412146620280340569649349251249)
sage: three_squares(11^111-1)
(616274160655975340150706442680, 901582938385735143295060746161, 6270382387635744140394001363065
sage: three_squares(7 * 2^41)
(1048576, 2097152, 3145728)
sage: three_squares(7 * 2^42)
Traceback (most recent call last):
...
ValueError: 30786325577728 is not a sum of 3 squares
sage: three_squares(0)
(0, 0, 0)
sage: three_squares(-1)
Traceback (most recent call last):
...
ValueError: -1 is not a sum of 3 squares
```

TESTS:

```
sage: for _ in xrange(100):
....:     a = ZZ.random_element(2**16, 2**20)
....:     b = ZZ.random_element(2**16, 2**20)
....:     c = ZZ.random_element(2**16, 2**20)
....:     n = a**2 + b**2 + c**2
....:     aa,bb,cc = three_squares(n)
....:     assert aa**2 + bb**2 + cc**2 == n
```

ALGORITHM:

See http://www.schorn.ch/howto.html

`sage.rings.arith.`**`trial_division`**(*n*, *bound=None*)

Return the smallest prime divisor <= bound of the positive integer n, or n if there is no such prime. If the optional argument bound is omitted, then bound <= n.

INPUT:

- `n` - a positive integer
- `bound` - (optional) a positive integer

OUTPUT:

- `int` - a prime p=bound that divides n, or n if there is no such prime.

EXAMPLES:

```
sage: trial_division(15)
3
sage: trial_division(91)
7
sage: trial_division(11)
```

```
11
sage: trial_division(387833, 300)
387833
sage: # 300 is not big enough to split off a
sage: # factor, but 400 is.
sage: trial_division(387833, 400)
389
```

sage.rings.arith.**two_squares**(*n*)

Write the integer n as a sum of two integer squares if possible; otherwise raise a ValueError.

INPUT:

- n – an integer

OUTPUT: a tuple (a, b) of non-negative integers such that $n = a^2 + b^2$ with $a <= b$.

EXAMPLES:

```
sage: two_squares(389)
(10, 17)
sage: two_squares(21)
Traceback (most recent call last):
...
ValueError: 21 is not a sum of 2 squares
sage: two_squares(21^2)
(0, 21)
sage: a,b = two_squares(100000000000000000129); a,b
(4418521500, 8970878873)
sage: a^2 + b^2
100000000000000000129
sage: two_squares(2^222+1)
(253801659504708621991421712450521, 2583712713213354898490304645018692)
sage: two_squares(0)
(0, 0)
sage: two_squares(-1)
Traceback (most recent call last):
...
ValueError: -1 is not a sum of 2 squares
```

TESTS:

```
sage: for _ in xrange(100):
....:     a = ZZ.random_element(2**16, 2**20)
....:     b = ZZ.random_element(2**16, 2**20)
....:     n = a**2 + b**2
....:     aa,bb = two_squares(n)
....:     assert aa**2 + bb**2 == n
```

ALGORITHM:

See http://www.schorn.ch/howto.html

sage.rings.arith.**valuation**(*m*, **args*, ***kwds*)

Return the valuation of m.

This function simply calls the m.valuation() method. See the documentation of m.valuation() for a more precise description.

Note that the use of this functions is discouraged as it is better to use m.valuation() directly.

Note: This is not always a valuation in the mathematical sense. For more information see: sage.rings.finite_rings.integer_mod.IntegerMod_int.valuation

EXAMPLES:

```
sage: valuation(512,2)
9
sage: valuation(1,2)
0
sage: valuation(5/9, 3)
-2
```

Valuation of 0 is defined, but valuation with respect to 0 is not:

```
sage: valuation(0,7)
+Infinity
sage: valuation(3,0)
Traceback (most recent call last):
...
ValueError: You can only compute the valuation with respect to a integer larger than 1.
```

Here are some other examples:

```
sage: valuation(100,10)
2
sage: valuation(200,10)
2
sage: valuation(243,3)
5
sage: valuation(243*10007,3)
5
sage: valuation(243*10007,10007)
1
sage: y = QQ['y'].gen()
sage: valuation(y^3, y)
3
sage: x = QQ[['x']].gen()
sage: valuation((x^3-x^2)/(x-4))
2
sage: valuation(4r,2r)
2
sage: valuation(1r,1r)
Traceback (most recent call last):
...
ValueError: You can only compute the valuation with respect to a integer larger than 1.
```

sage.rings.arith.**xgcd**(a, b)

Return a triple `(g,s,t)` such that $g = s \cdot a + t \cdot b = \gcd(a, b)$.

Note: One exception is if a and b are not in a principal ideal domain (see Wikipedia article Principal_ideal_domain), e.g., they are both polynomials over the integers. Then this function can't in general return `(g,s,t)` as above, since they need not exist. Instead, over the integers, we first multiply g by a divisor of the resultant of a/g and b/g, up to sign.

INPUT:

- `a, b` - integers or more generally, element of a ring for which the xgcd make sense (e.g. a field or univariate polynomials).

OUTPUT:

- g, s, t - such that $g = s \cdot a + t \cdot b$

Note: There is no guarantee that the returned cofactors (s and t) are minimal.

EXAMPLES:

```
sage: xgcd(56, 44)
(4, 4, -5)
sage: 4*56 + (-5)*44
4

sage: g, a, b = xgcd(5/1, 7/1); g, a, b
(1, 3, -2)
sage: a*(5/1) + b*(7/1) == g
True

sage: x = polygen(QQ)
sage: xgcd(x^3 - 1, x^2 - 1)
(x - 1, 1, -x)

sage: K.<g> = NumberField(x^2-3)
sage: g.xgcd(g+2)
(1, 1/3*g, 0)

sage: R.<a,b> = K[]
sage: S.<y> = R.fraction_field()[]
sage: xgcd(y^2, a*y+b)
(1, a^2/b^2, ((-a)/b^2)*y + 1/b)
sage: xgcd((b+g)*y^2, (a+g)*y+b)
(1, (a^2 + (2*g)*a + 3)/(b^3 + (g)*b^2), ((-a + (-g))/b^2)*y + 1/b)
```

Here is an example of a xgcd for two polynomials over the integers, where the linear combination is not the gcd but the gcd multiplied by the resultant:

```
sage: R.<x> = ZZ[]
sage: gcd(2*x*(x-1), x^2)
x
sage: xgcd(2*x*(x-1), x^2)
(2*x, -1, 2)
sage: (2*(x-1)).resultant(x)
2
```

`sage.rings.arith.`**`xkcd`**`(n='')`

This function is similar to the xgcd function, but behaves in a completely different way.

INPUT:

- n - an integer (optional)

OUTPUT:

This function outputs nothing it just prints something. Note that this function does not feel itself at ease in a html deprived environment.

EXAMPLES:

```
sage: xkcd(353) # optional - internet
<html><font color='black'><h1>Python</h1><img src="http://imgs.xkcd.com/comics/python.png" title
```

sage.rings.arith.**xlcm**(*m*, *n*)

Extended lcm function: given two positive integers m, n, returns a triple (l, m_1, n_1) such that $l = \mathrm{lcm}(m, n) = m_1 \cdot n_1$ where $m_1|m$, $n_1|n$ and $\gcd(m_1, n_1) = 1$, all with no factorization.

Used to construct an element of order l from elements of orders m, n in any group: see sage/groups/generic.py for examples.

EXAMPLES:

```
sage: xlcm(120,36)
(360, 40, 9)
```

CHAPTER

THIRTYSIX

IMPLEMENT FAST VERSION OF DECOMPOSITION OF (SMALL) INTEGERS INTO SUM OF SQUARES

by direct method not relying on factorisation.

AUTHORS:

- Vincent Delecroix (2014): first implementation (trac ticket #16374)

sage.rings.sum_of_squares.**four_squares_pyx**(n)

Return a 4-tuple of non-negative integers `(i,j,k,l)` such that $i^2 + j^2 + k^2 + l^2 = n$ and $i \leq j \leq k \leq l$.

The input must be lesser than $2^{32} = 4294967296$, otherwise an `OverflowError` is raised.

See also:

`four_squares()` is much more suited for large input

EXAMPLES:

```
sage: from sage.rings.sum_of_squares import four_squares_pyx
sage: four_squares_pyx(15447)
(2, 5, 17, 123)
sage: 2^2 + 5^2 + 17^2 + 123^2
15447

sage: four_squares_pyx(523439)
(3, 5, 26, 723)
sage: 3^2 + 5^2 + 26^2 + 723^2
523439

sage: four_squares_pyx(2**32)
Traceback (most recent call last):
...
OverflowError: ...
```

TESTS:

```
sage: four_squares_pyx(0)
(0, 0, 0, 0)

sage: s = lambda (x,y,z,t): x**2 + y**2 + z**2 + t**2
sage: all(s(four_squares_pyx(n)) == n for n in xrange(5000,10000))
True
```

sage.rings.sum_of_squares.**is_sum_of_two_squares_pyx**(n)

Return `True` if `n` is a sum of two squares and `False` otherwise.

The input must be smaller than $2^{32} = 4294967296$, otherwise an `OverflowError` is raised.

EXAMPLES:

```
sage: from sage.rings.sum_of_squares import is_sum_of_two_squares_pyx
sage: filter(is_sum_of_two_squares_pyx, range(30))
[0, 1, 2, 4, 5, 8, 9, 10, 13, 16, 17, 18, 20, 25, 26, 29]

sage: is_sum_of_two_squares_pyx(2**32)
Traceback (most recent call last):
...
OverflowError: ...
```

sage.rings.sum_of_squares.**three_squares_pyx**(*n*)

If n is a sum of three squares return a 3-tuple (i,j,k) of Sage integers such that $i^2 + j^2 + k^2 = n$ and $i \leq j \leq k$. Otherwise raise a ValueError.

The input must be lesser than $2^{32} = 4294967296$, otherwise an OverflowError is raised.

EXAMPLES:

```
sage: from sage.rings.sum_of_squares import three_squares_pyx
sage: three_squares_pyx(0)
(0, 0, 0)
sage: three_squares_pyx(1)
(0, 0, 1)
sage: three_squares_pyx(2)
(0, 1, 1)
sage: three_squares_pyx(3)
(1, 1, 1)
sage: three_squares_pyx(4)
(0, 0, 2)
sage: three_squares_pyx(5)
(0, 1, 2)
sage: three_squares_pyx(6)
(1, 1, 2)
sage: three_squares_pyx(7)
Traceback (most recent call last):
...
ValueError: 7 is not a sum of 3 squares
sage: three_squares_pyx(107)
(1, 5, 9)

sage: three_squares_pyx(2**32)
Traceback (most recent call last):
...
OverflowError: ...
```

TESTS:

```
sage: s = lambda (x,y,z) : x**2 + y**2 + z**2
sage: for ijk in Subsets(Subsets(35000,15).random_element(),3):
....:     if s(three_squares_pyx(s(ijk))) != s(ijk):
....:         print "hey"
```

sage.rings.sum_of_squares.**two_squares_pyx**(*n*)

Return a pair of non-negative integers (i,j) such that $i^2 + j^2 = n$.

If n is not a sum of two squares, a ValueError is raised. The input must be lesser than $2^{32} = 4294967296$, otherwise an OverflowError is raised.

See also:

`two_squares()` is much more suited for large inputs

EXAMPLES:

```
sage: from sage.rings.sum_of_squares import two_squares_pyx
sage: two_squares_pyx(0)
(0, 0)
sage: two_squares_pyx(1)
(0, 1)
sage: two_squares_pyx(2)
(1, 1)
sage: two_squares_pyx(3)
Traceback (most recent call last):
...
ValueError: 3 is not a sum of 2 squares
sage: two_squares_pyx(106)
(5, 9)

sage: two_squares_pyx(2**32)
Traceback (most recent call last):
...
OverflowError: ...
```

TESTS:

```
sage: s = lambda (x,y) : x**2 + y**2
sage: for ij in Subsets(Subsets(45000,15).random_element(),2):
....:     if s(two_squares_pyx(s(ij))) != s(ij):
....:         print "hey"

sage: for n in xrange(1,65536):
....:     if two_squares_pyx(n**2) != (0, n):
....:         print "hey"
....:     if two_squares_pyx(n**2+1) != (1, n):
....:         print "ho"
```

CHAPTER

THIRTYSEVEN

FIXING PICKLE FOR NESTED CLASSES

As of Python 2.6, names for nested classes are set by Python in a way which is incompatible with the pickling of such classes (pickling by name):

```
sage: class A:
...       class B:
...           pass
sage: A.B.__name__
'B'
```

instead of the a priori more natural `"A.B"`.

Furthermore, upon pickling (here in save_global) *and* unpickling (in load_global) a class with name `"A.B"` in a module `mod`, the standard cPickle module searches for `"A.B"` in `mod.__dict__` instead of looking up `"A"` and then `"B"` in the result.

See: http://groups.google.com/group/sage-devel/browse_thread/thread/6c7055f4a580b7ae/

This module provides two utilities to workaround this issue:

- `nested_pickle()` "fixes" recursively the name of the subclasses of a class and inserts their fullname `"A.B"` in `mod.__dict__`
- `NestedClassMetaclass` is a metaclass ensuring that `nested_pickle()` is called on a class upon creation.

See also `sage.misc.nested_class_test`.

EXAMPLES:

```
sage: from sage.misc.nested_class import A1, nested_pickle

sage: A1.A2.A3.__name__
'A3'
sage: A1.A2.A3
<class sage.misc.nested_class.A3 at ...>

sage: nested_pickle(A1)
<class sage.misc.nested_class.A1 at ...>

sage: A1.A2
<class sage.misc.nested_class.A1.A2 at ...>

sage: A1.A2.A3
<class sage.misc.nested_class.A1.A2.A3 at ...>
sage: A1.A2.A3.__name__
'A1.A2.A3'
```

```
sage: sage.misc.nested_class.__dict__['A1.A2'] is A1.A2
True
sage: sage.misc.nested_class.__dict__['A1.A2.A3'] is A1.A2.A3
True
```

All of this is not perfect. In the following scenario:

```
sage: class A1:
...       class A2:
...           pass
sage: class B1:
...       A2 = A1.A2
...
```

The name for "A1.A2" could potentially be set to "B1.A2". But that will work anyway.

`sage.misc.nested_class.`**`modify_for_nested_pickle`**(*cls*, *name_prefix*, *module*, *first_run=True*)

Modify the subclasses of the given class to be picklable, by giving them a mangled name and putting the mangled name in the module namespace.

INPUTS:

- `cls` - The class to modify.
- `name_prefix` - The prefix to prepend to the class name.
- `module` - The module object to modify with the mangled name.
- `first_run` - optional bool (default True): Whether or not this function is run for the first time on `cls`.

NOTE:

This function would usually not be directly called. It is internally used in `NestedClassMetaclass`.

EXAMPLES:

```
sage: from sage.misc.nested_class import *
sage: class A(object):
...       class B(object):
...           pass
...
sage: module = sys.modules['__main__']
sage: A.B.__name__
'B'
sage: getattr(module, 'A.B', 'Not found')
'Not found'
sage: modify_for_nested_pickle(A, 'A', sys.modules['__main__'])
sage: A.B.__name__
'A.B'
sage: getattr(module, 'A.B', 'Not found')
<class '__main__.A.B'>
```

Here we demonstrate the effect of the `first_run` argument:

```
sage: modify_for_nested_pickle(A, 'X', sys.modules['__main__'])
sage: A.B.__name__ # nothing changed
'A.B'
sage: modify_for_nested_pickle(A, 'X', sys.modules['__main__'], first_run=False)
sage: A.B.__name__
'X.A.B'
```

Note that the class is now found in the module under both its old and its new name:

```
sage: getattr(module, 'A.B', 'Not found')
<class '__main__.X.A.B'>
sage: getattr(module, 'X.A.B', 'Not found')
<class '__main__.X.A.B'>
```

TESTS:

The following is a real life example, that was enabled by the internal use of the``first_run`` in trac ticket #9107:

```
sage: cython_code = [
....:  "from sage.structure.unique_representation import UniqueRepresentation",
....:  "class A1(UniqueRepresentation):",
....:  "    class B1(UniqueRepresentation):",
....:  "        class C1: pass",
....:  "    class B2:",
....:  "        class C2: pass"]
sage: import os
sage: cython(os.linesep.join(cython_code))
```

Before trac ticket #9107, the name of `A1.B1.C1` would have been wrong:

```
sage: A1.B1.C1.__name__
'A1.B1.C1'
sage: A1.B2.C2.__name__
'A1.B2.C2'
sage: A_module = sys.modules[A1.__module__]
sage: getattr(A_module, 'A1.B1.C1', 'Not found').__name__
'A1.B1.C1'
sage: getattr(A_module, 'A1.B2.C2', 'Not found').__name__
'A1.B2.C2'
```

`sage.misc.nested_class.`**`nested_pickle`**`(`*cls*`)`

This decorator takes a class that potentially contains nested classes. For each such nested class, its name is modified to a new illegal identifier, and that name is set in the module. For example, if you have:

```
sage: from sage.misc.nested_class import nested_pickle
sage: module = sys.modules['__main__']
sage: class A(object):
...       class B:
...           pass
sage: nested_pickle(A)
<class '__main__.A'>
```

then the name of class `"B"` will be modified to `"A.B"`, and the `"A.B"` attribute of the module will be set to class `"B"`:

```
sage: A.B.__name__
'A.B'
sage: getattr(module, 'A.B', 'Not found')
<class __main__.A.B at ...>
```

In Python 2.6, decorators work with classes; then `@nested_pickle` should work as a decorator:

```
sage: @nested_pickle    # todo: not implemented
...   class A2(object):
...       class B:
...           pass
sage: A2.B.__name__    # todo: not implemented
'A2.B'
```

```
sage: getattr(module, 'A2.B', 'Not found')     # todo: not implemented
<class __main__.A2.B at ...>
```

EXAMPLES:

```
sage: from sage.misc.nested_class import *
sage: loads(dumps(MainClass.NestedClass())) # indirect doctest
<sage.misc.nested_class.MainClass.NestedClass object at 0x...>
```

class sage.misc.nested_class.**NestedClassMetaclass**

Bases: type

A metaclass for nested pickling.

Check that one can use a metaclass to ensure nested_pickle is called on any derived subclass:

```
sage: from sage.misc.nested_class import NestedClassMetaclass
sage: class ASuperClass(object):
...       __metaclass__ = NestedClassMetaclass
...
sage: class A3(ASuperClass):
...       class B(object):
...           pass
...
sage: A3.B.__name__
'A3.B'
sage: getattr(sys.modules['__main__'], 'A3.B', 'Not found')
<class '__main__.A3.B'>
```

class sage.misc.nested_class.**MainClass**

Bases: object

A simple class to test nested_pickle.

EXAMPLES:

```
sage: from sage.misc.nested_class import *
sage: loads(dumps(MainClass()))
<sage.misc.nested_class.MainClass object at 0x...>
```

class NestedClass

Bases: object

EXAMPLES:

```
sage: from sage.misc.nested_class import *
sage: loads(dumps(MainClass.NestedClass()))
<sage.misc.nested_class.MainClass.NestedClass object at 0x...>
```

class NestedSubClass

Bases: object

EXAMPLES:

```
sage: from sage.misc.nested_class import *
sage: loads(dumps(MainClass.NestedClass.NestedSubClass()))
<sage.misc.nested_class.MainClass.NestedClass.NestedSubClass object at 0x...>
sage: getattr(sage.misc.nested_class, 'MainClass.NestedClass.NestedSubClass')
<class 'sage.misc.nested_class.MainClass.NestedClass.NestedSubClass'>
sage: MainClass.NestedClass.NestedSubClass.__name__
'MainClass.NestedClass.NestedSubClass'
```

CHAPTER

THIRTYEIGHT

TEST FOR NESTED CLASS PARENT

This file contains a discussion, examples, and tests about nested classes and parents. It is kept in a separate file to avoid import loops.

EXAMPLES:

Currently pickling fails for parents using nested classes (typically for categories), but deriving only from Parent:

```
sage: from sage.misc.nested_class_test import TestParent1, TestParent2, TestParent3, TestParent4
sage: P = TestParent1()
sage: TestSuite(P).run()
Failure ...
The following tests failed: _test_elements, _test_pickling
```

They actually need to be in the NestedClassMetaclass. However, due to a technical detail, this is currently not directly supported:

```
sage: P = TestParent2()
Traceback (most recent call last):
...
TypeError: metaclass conflict: the metaclass of a derived class must be a (non-strict) subclass of th
sage: TestSuite(P).run()  # not tested
```

Instead, the easiest is to inherit from UniqueRepresentation, which is what you want to do anyway most of the time:

```
sage: P = TestParent3()
sage: TestSuite(P).run()
```

This is what all Sage's parents using categories currently do. An alternative is to use ClasscallMetaclass as metaclass:

```
sage: P = TestParent4()
sage: TestSuite(P).run()
```

CHAPTER

THIRTYNINE

SPECIAL METHODS FOR CLASSES

AUTHORS:

- Nicolas M. Thiery (2009-2011) implementation of `__classcall__`, `__classget__`, `__classcontains__`;
- Florent Hivert (2010-2012): implementation of `__classcall_private__`, documentation, Cythonization and optimization.

class `sage.misc.classcall_metaclass.`**`ClasscallMetaclass`**

Bases: `sage.misc.nested_class.NestedClassMetaclass`

A metaclass providing support for special methods for classes.

From the Section Special method names of the Python Reference Manual:

> 'a class `cls` can implement certain operations on its instances that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names'.

The purpose of this metaclass is to allow for the class `cls` to implement analogues of those special methods for the operations on the class itself.

Currently, the following special methods are supported:

- `.__classcall__` (and `.__classcall_private__`) for customizing `cls(...)` (analogue of `.__call__`).
- `.__classcontains__` for customizing membership testing `x in cls` (analogue of `.__contains__`).
- `.__classget__` for customizing the binding behavior in `foo.cls` (analogue of `.__get__`).

See the documentation of `__call__()` and of `__get__()` and `__contains__()` for the description of the respective protocols.

> **Warning:** For technical reasons, `__classcall__`, `__classcall_private__`, `__classcontains__`, and `__classget__` must be defined as `staticmethod()`'s, even though they receive the class itself as their first argument.

> **Warning:** For efficiency reasons, the resolution for the special methods is done once for all, upon creation of the class. Thus, later dynamic changes to those methods are ignored. But see also `_set_classcall()`.

`ClasscallMetaclass` is an extension of the base `type`.

TODO: find a good name for this metaclass.

TESTS:

```
sage: PerfectMatchings(2).list()
[[(1, 2)]]
```

Note: If a class is put in this metaclass it automatically becomes a new-style class:

```
sage: from sage.misc.classcall_metaclass import ClasscallMetaclass
sage: class Foo:
...       __metaclass__ = ClasscallMetaclass
sage: x = Foo(); x
<__main__.Foo object at 0x...>
sage: issubclass(Foo, object)
True
sage: isinstance(Foo, type)
True
```

sage.misc.classcall_metaclass.**typecall**(*cls*, **args*, ***opts*)

Object construction

This is a faster equivalent to `type.__call__(cls, <some arguments>)`.

INPUT:

- `cls` – the class used for constructing the instance. It must be a builtin type or a new style class (inheriting from `object`).

EXAMPLES:

```
sage: from sage.misc.classcall_metaclass import typecall
sage: class Foo(object): pass
sage: typecall(Foo)
<__main__.Foo object at 0x...>
sage: typecall(list)
[]
sage: typecall(Integer, 2)
2
```

Warning: `typecall()` doesn't work for old style class (not inheriting from `object`):

```
sage: class Bar: pass
sage: typecall(Bar)
Traceback (most recent call last):
...
TypeError: Argument 'cls' has incorrect type (expected type, got classobj)
```

sage.misc.classcall_metaclass.**timeCall**(*T*, *n*, **args*)

We illustrate some timing when using the classcall mechanism.

EXAMPLES:

```
sage: from sage.misc.classcall_metaclass import (
...       ClasscallMetaclass, CRef, C2, C3, C2C, timeCall)
sage: timeCall(object, 1000)
```

For reference let construct basic objects and a basic Python class:

```
sage: %timeit timeCall(object, 1000)   # not tested
625 loops, best of 3: 41.4 µs per loop
```

```
sage: i1 = int(1); i3 = int(3) # don't use Sage's Integer
sage: class PRef(object):
...       def __init__(self, i):
...           self.i = i+i1
```

For a Python class, compared to the reference class there is a 10% overhead in using `ClasscallMetaclass` if there is no classcall defined:

```
sage: class P(object):
...       __metaclass__ = ClasscallMetaclass
...       def __init__(self, i):
...           self.i = i+i1

sage: %timeit timeCall(PRef, 1000, i3)   # not tested
625 loops, best of 3: 420 µs per loop
sage: %timeit timeCall(P, 1000, i3)      # not tested
625 loops, best of 3: 458 µs per loop
```

For a Cython class (not cdef since they doesn't allows metaclasses), the overhead is a little larger:

```
sage: %timeit timeCall(CRef, 1000, i3)   # not tested
625 loops, best of 3: 266 µs per loop
sage: %timeit timeCall(C2, 1000, i3)     # not tested
625 loops, best of 3: 298 µs per loop
```

Let's now compare when there is a classcall defined:

```
sage: class PC(object):
...       __metaclass__ = ClasscallMetaclass
...       @staticmethod
...       def __classcall__(cls, i):
...           return i+i1
sage: %timeit timeCall(C2C, 1000, i3)   # not tested
625 loops, best of 3: 148 µs per loop
sage: %timeit timeCall(PC, 1000, i3)    # not tested
625 loops, best of 3: 289 µs per loop
```

The overhead of the indirection (`C(...) -> ClasscallMetaclass.__call__(...) -> C.__classcall__(...)`) is unfortunately quite large in this case (two method calls instead of one). In reasonable usecases, the overhead should be mostly hidden by the computations inside the classcall:

```
sage: %timeit timeCall(C2C.__classcall__, 1000, C2C, i3)  # not tested
625 loops, best of 3: 33 µs per loop
sage: %timeit timeCall(PC.__classcall__, 1000, PC, i3)    # not tested
625 loops, best of 3: 131 µs per loop
```

Finally, there is no significant difference between Cython's V2 and V3 syntax for metaclass:

```
sage: %timeit timeCall(C2, 1000, i3)   # not tested
625 loops, best of 3: 330 µs per loop
sage: %timeit timeCall(C3, 1000, i3)   # not tested
625 loops, best of 3: 328 µs per loop
```

CHAPTER

FORTY

FAST METHODS VIA CYTHON

This module provides extension classes with useful methods of cython speed, that python classes can inherit.

Note: This module provides a cython base class WithEqualityById implementing unique instance behaviour, and a cython base class FastHashable_class, which has a quite fast hash whose value can be freely chosen at initialisation time.

AUTHOR:

- Simon King (2013-02): Original version
- Simon King (2013-10): Add Singleton

class sage.misc.fast_methods.**FastHashable_class**

Bases: object

A class that has a fast hash method, returning a pre-assigned value.

NOTE:

This is for internal use only. The class has a cdef attribute _hash, that needs to be assigned (for example, by calling the init method, or by a direct assignement using cython). This is slower than using provide_hash_by_id(), but has the advantage that the hash can be prescribed, by assigning a cdef attribute _hash.

TESTS:

```
sage: from sage.misc.fast_methods import FastHashable_class
sage: H = FastHashable_class(123)
sage: hash(H)
123
```

class sage.misc.fast_methods.**Singleton**

Bases: sage.misc.fast_methods.WithEqualityById

A base class for singletons.

A singleton is a class that allows to create not more than a single instance. This instance can also belong to a subclass, but it is not possible to have several subclasses of a singleton all having distinct unique instances.

In order to create a singleton, just add Singleton to the list of base classes:

```
sage: from sage.misc.fast_methods import Singleton
sage: class C(Singleton, SageObject):
....:     def __init__(self):
....:         print "creating singleton"
sage: c = C()
creating singleton
sage: c2 = C()
```

```
sage: c is c2
True
```

The unique instance of a singleton stays in memory as long as the singleton itself does.

Pickling, copying, hashing, and comparison are provided for by `Singleton` according to the singleton paradigm. Note that pickling fails if the class is replaced by a sub-sub-class after creation of the instance:

```
sage: class D(C):
....:     pass
sage: import __main__      # This is only needed ...
sage: __main__.C = C       # ... in doctests
sage: __main__.D = D       # same here, only in doctests
sage: orig = type(c)
sage: c.__class__ = D
sage: orig == type(c)
False
sage: loads(dumps(c))
Traceback (most recent call last):
...
AssertionError: (("<class '__main__.D'> is not a direct
subclass of <class 'sage.misc.fast_methods.Singleton'>",),
<class '__main__.D'>, ())
```

class `sage.misc.fast_methods.`**`WithEqualityById`**

Bases: `object`

Provide hash and equality test based on identity.

Note: This class provides the unique representation behaviour of `UniqueRepresentation`, together with `CachedRepresentation`.

EXAMPLES:

Any instance of `UniqueRepresentation` inherits from `WithEqualityById`.

```
sage: class MyParent(Parent):
...     def __init__(self, x):
...         self.x = x
...     def __cmp__(self,other):
...         return cmp(self.x^2,other.x^2)
...     def __hash__(self):
...         return hash(self.x)
sage: class MyUniqueParent(UniqueRepresentation, MyParent): pass
sage: issubclass(MyUniqueParent, sage.misc.fast_methods.WithEqualityById)
True
```

Inheriting from `WithEqualityById` provides unique representation behaviour. In particular, the comparison inherited from `MyParent` is overloaded:

```
sage: a = MyUniqueParent(1)
sage: b = MyUniqueParent(2)
sage: c = MyUniqueParent(1)
sage: a is c
True
sage: d = MyUniqueParent(-1)
sage: a == d
False
```

Note, however, that Python distinguishes between "comparison by cmp" and "comparison by binary relations":

```
sage: cmp(a,d)
0
```

The comparison inherited from `MyParent` will be used in those cases in which identity does not give sufficient information to find the relation:

```
sage: a < b
True
sage: b > d
True
```

The hash inherited from `MyParent` is replaced by a hash that coincides with `object`'s hash:

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

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

```
sage: class MyNonUniqueParent(MyUniqueParent):
...     def __eq__(self, other):
...         return self.x^2 == other.x^2
sage: a = MyNonUniqueParent(1)
sage: d = MyNonUniqueParent(-1)
sage: a is MyNonUniqueParent(1)
True
sage: a == d
True
sage: a is d
False
```

CHAPTER FORTYONE

UNIT TESTING FOR SAGE OBJECTS

class sage.misc.sage_unittest.**InstanceTester**(*instance*, *elements=None*, *verbose=False*, *prefix=''*, *max_runs=4096*, ***options*)

Bases: unittest.case.TestCase

A gadget attached to an instance providing it with testing utilities.

EXAMPLES:

```
sage: from sage.misc.sage_unittest import InstanceTester
sage: InstanceTester(instance = ZZ, verbose = True, elements = [1,2,3])
Testing utilities for Integer Ring
```

This is used by SageObject._tester, which see:

```
sage: QQ._tester()
Testing utilities for Rational Field
```

info(*message*, *newline=True*)

Displays user information

EXAMPLES:

```
sage: from sage.misc.sage_unittest import InstanceTester
sage: tester = InstanceTester(ZZ, verbose = True)

sage: tester.info("hello"); tester.info("world")
hello
world

sage: tester = InstanceTester(ZZ, verbose = False)
sage: tester.info("hello"); tester.info("world")

sage: tester = InstanceTester(ZZ, verbose = True)
sage: tester.info("hello", newline = False); tester.info(" world")
hello world
```

runTest()

Trivial implementation of unittest.TestCase.runTest() to please the super class TestCase. That's the price to pay for abusively inheriting from it.

EXAMPLES:

```
sage: from sage.misc.sage_unittest import InstanceTester
sage: tester = InstanceTester(ZZ, verbose = True)
sage: tester.runTest()
```

some_elements(*S=None*)

Returns a list (or iterable) of elements of `self` on which the tests should be run. This is only meaningful for container objects like parents.

INPUT:

- `S` – a set of elements to select from. By default this will use the elements passed to this tester at creation time, or the result of `some_elements()` if no elements were specified.

OUTPUT:

A list of at most `self._max_runs` elements of `S`.

EXAMPLES:

By default, this calls `some_elements()` on the instance:

```
sage: from sage.misc.sage_unittest import InstanceTester
sage: class MyParent(Parent):
...       def some_elements(self):
...           return [1,2,3,4,5]
...
sage: tester = InstanceTester(MyParent())
sage: list(tester.some_elements())
[1, 2, 3, 4, 5]

sage: tester = InstanceTester(MyParent(), max_runs=3)
sage: list(tester.some_elements())
[1, 2, 3]

sage: tester = InstanceTester(MyParent(), max_runs=7)
sage: list(tester.some_elements())
[1, 2, 3, 4, 5]

sage: tester = InstanceTester(MyParent(), elements=[1,3,5])
sage: list(tester.some_elements())
[1, 3, 5]

sage: tester = InstanceTester(MyParent(), elements=[1,3,5], max_runs=2)
sage: list(tester.some_elements())
[1, 3]

sage: tester = InstanceTester(FiniteEnumeratedSet(['a','b','c','d']), max_runs=3)
sage: tester.some_elements()
['a', 'b', 'c']

sage: tester = InstanceTester(FiniteEnumeratedSet([]))
sage: list(tester.some_elements())
[]

sage: tester = InstanceTester(ZZ)
sage: ZZ.some_elements()              # yikes, shamelessly trivial ...
<generator object _some_elements_from_iterator at 0x...>
sage: list(tester.some_elements())
[0, 1, -1, 2, -2, ..., 49, -49, 50]

sage: tester = InstanceTester(ZZ, elements = ZZ, max_runs=5)
sage: list(tester.some_elements())
[0, 1, -1, 2, -2]

sage: tester = InstanceTester(ZZ, elements = srange(100), max_runs=5)
```

```
sage: list(tester.some_elements())
[0, 1, 2, 3, 4]

sage: tester = InstanceTester(ZZ, elements = srange(3), max_runs=5)
sage: list(tester.some_elements())
[0, 1, 2]
```

Test for trac ticket #15919, trac ticket #16244:

```
sage: Z = IntegerModRing(25) # random.sample, which was used pre #16244, has a threshold at
sage: Z[1]                   # since #8389, indexed access is used for ring extensions
Traceback (most recent call last):
...
ValueError: first letter of variable name must be a letter
sage: tester = InstanceTester(Z, elements=Z, max_runs=5)
sage: list(tester.some_elements())
[0, 1, 2, 3, 4]

sage: C = CartesianProduct(Z, Z, Z, Z)
sage: len(C)
390625
sage: tester = InstanceTester(C, elements = C, max_runs=4)
sage: list(tester.some_elements())
[[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 0, 2], [0, 0, 0, 3]]
```

class `sage.misc.sage_unittest.`**`PythonObjectWithTests`**(*instance*)

Bases: `object`

Utility class for running basis tests on a plain Python object (that is not in SageObject). More test methods can be added here.

EXAMPLES:

```
sage: TestSuite("bla").run()
```

class `sage.misc.sage_unittest.`**`TestSuite`**(*instance*)

Bases: `object`

Test suites for Sage objects.

EXAMPLES:

```
sage: TestSuite(ZZ).run()
```

No output means that all tests passed. Which tests? In practice this calls all the methods `._test_*` of this object, in alphabetic order:

```
sage: TestSuite(1).run(verbose = True)
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_nonzero_equal() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
```

Those methods are typically implemented by abstract super classes, in particular via categories, in order to enforce standard behavior and API, or provide mathematical sanity checks. For example if `self` is in the category of finite semigroups, this checks that the multiplication is associative (at least on some elements):

```
sage: S = FiniteSemigroups().example(alphabet = ('a', 'b'))
sage: TestSuite(S).run(verbose = True)
running ._test_an_element() . . . pass
```

```
running ._test_associativity() . . . pass
running ._test_category() . . . pass
running ._test_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_enumerated_set_contains() . . . pass
running ._test_enumerated_set_iter_cardinality() . . . pass
running ._test_enumerated_set_iter_list() . . . pass
running ._test_eq() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

The different test methods can be called independently:

```
sage: S._test_associativity()
```

Debugging tip: in case of failure of some test, use `%pdb on` to turn on automatic debugging on error. Run the failing test independtly: the debugger will stop right where the first assertion fails. Then, introspection can be used to analyse what exactly the problem is. See also the `catch = False` option to `run()`.

When meaningful, one can further customize on which elements the tests are run. Here, we use it to *prove* that the multiplication is indeed associative, by running the test on all the elements:

```
sage: S._test_associativity(elements = S)
```

Adding a new test boils down to adding a new method in the class of the object or any super class (e.g. in a category). This method should use the utility `_tester()` to handle standard options and report test failures. See the code of `_test_an_element()` for an example. Note: Python's testunit convention is to look for methods called `.test*`; we use instead `._test_*` so as not to pollute the object's interface.

Eventually, every implementation of a `SageObject` should run a `TestSuite` on one of its instances in its doctest (replacing the current `loads(dumps(x))` tests).

Finally, running `TestSuite` on a standard Python object does some basic sanity checks:

```
sage: TestSuite(int(1)).run(verbose = True)
running ._test_pickling() . . . pass
```

TODO:

- Allow for customized behavior in case of failing assertion (warning, error, statistic accounting). This involves reimplementing the methods fail / failIf / ... of unittest.TestCase in InstanceTester
- Don't catch the exceptions if `TestSuite(..).run()` is called under the debugger, or with `%pdb` on (how to detect this? see `get_ipython()`, `IPython.Magic.shell.call_pdb`, ...) In the mean time, see the `catch=False` option.
- Run the tests according to the inheritance order, from most generic to most specific, rather than alphabetically. Then, the first failure will be the most relevant, the others being usually consequences.
- Improve integration with doctests (statistics on failing/passing tests)

- Add proper support for nested testsuites.
- Integration with unittest: Make TestSuite inherit from unittest.TestSuite? Make `.run(...)` accept a result object
- Add some standard option `proof = True`, asking for the test method to choose appropriately the elements so as to prove the desired property. The test method may assume that a parent implements properly all the super categories. For example, the `_test_commutative` method of the category `CommutativeSemigroups()` may just check that the provided generators commute, implicitly assuming that generators indeed generate the semigroup (as required by `Semigroups()`).

run (*category=None*, *skip=[ ]*, *catch=True*, *raise_on_failure=False*, ***options*)
Run all the tests from this test suite:

INPUT:

- `category` - a category; reserved for future use
- `skip` - a string or list (or iterable) of strings
- `raise_on_failure` - a boolean (default: False)
- `catch` - a boolean (default: True)

All other options are passed down to the individual tests.

EXAMPLES:

```
sage: TestSuite(ZZ).run()
```

We now use the `verbose` option:

```
sage: TestSuite(1).run(verbose = True)
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_nonzero_equal() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
```

Some tests may be skipped using the `skip` option:

```
sage: TestSuite(1).run(verbose = True, skip ="_test_pickling")
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_nonzero_equal() . . . pass
running ._test_not_implemented_methods() . . . pass
sage: TestSuite(1).run(verbose = True, skip =["_test_pickling", "_test_category"])
running ._test_eq() . . . pass
running ._test_nonzero_equal() . . . pass
running ._test_not_implemented_methods() . . . pass
```

We now show (and test) some standard error reports:

```
sage: class Blah(SageObject):
...       def _test_a(self, tester): pass
...       def _test_b(self, tester): tester.fail()
...       def _test_c(self, tester): pass
...       def _test_d(self, tester): tester.fail()

sage: TestSuite(Blah()).run()
Failure in _test_b:
Traceback (most recent call last):
  ...
```

```
AssertionError: None
------------------------------------------------------------
Failure in _test_d:
Traceback (most recent call last):
  ...
AssertionError: None
------------------------------------------------------------
Failure in _test_pickling:
Traceback (most recent call last):
  ...
PicklingError: Can't pickle <class '__main__.Blah'>: attribute lookup __main__.Blah failed
------------------------------------------------------------
The following tests failed: _test_b, _test_d, _test_pickling

sage: TestSuite(Blah()).run(verbose = True)
running ._test_a() . . . pass
running ._test_b() . . . fail
Traceback (most recent call last):
  ...
AssertionError: None
------------------------------------------------------------
running ._test_c() . . . pass
running ._test_category() . . . pass
running ._test_d() . . . fail
Traceback (most recent call last):
  ...
AssertionError: None
------------------------------------------------------------
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . fail
Traceback (most recent call last):
  ...
PicklingError: Can't pickle <class '__main__.Blah'>: attribute lookup __main__.Blah failed
------------------------------------------------------------
The following tests failed: _test_b, _test_d, _test_pickling

File "/opt/sage/local/lib/python/site-packages/sage/misc/sage_unittest.py", line 183, in run
test_method(tester = tester)
```

The `catch=False` option prevents `TestSuite` from catching exceptions:

```
sage: TestSuite(Blah()).run(catch=False)
Traceback (most recent call last):
  ...
  File ..., in _test_b
    def _test_b(self, tester): tester.fail()
  ...
AssertionError: None
```

In conjonction with `%pdb on`, this allows for the debbuger to jump directly to the first failure location.

exception `sage.misc.sage_unittest.`**`TestSuiteFailure`**

Bases: `exceptions.AssertionError`

x.__init__(...) initializes x; see help(type(x)) for signature

`sage.misc.sage_unittest.`**`instance_tester`**(*instance*, *tester=None*, ***options*)

Returns a gadget attached to `instance` providing testing utilities.

EXAMPLES:

```
sage: from sage.misc.sage_unittest import instance_tester
sage: tester = instance_tester(ZZ)

sage: tester.assert_(1 == 1)
sage: tester.assert_(1 == 0)
Traceback (most recent call last):
...
AssertionError: False is not true
sage: tester.assert_(1 == 0, "this is expected to fail")
Traceback (most recent call last):
...
AssertionError: this is expected to fail

sage: tester.assertEquals(1, 1)
sage: tester.assertEquals(1, 0)
Traceback (most recent call last):
...
AssertionError: 1 != 0
```

The available assertion testing facilities are the same as in `unittest.TestCase` [UNITTEST], which see (actually, by a slight abuse, tester is currently an instance of this class).

TESTS:

```
sage: instance_tester(ZZ, tester = tester) is tester
True
```

REFERENCES:

CHAPTER

FORTYTWO

RANDOM NUMBER STATES

AUTHORS:

- Carl Witty (2008-03): new file

This module manages all the available pseudo-random number generators in Sage. (For the rest of the documentation in this module, we will drop the "pseudo".)

The goal is to allow algorithms using random numbers to be reproducible from one run of Sage to the next, and (to the extent possible) from one machine to the next (even across different operating systems and architectures).

There are two parts to the API. First we will describe the command line oriented API, for setting random number generator seeds. Then we will describe the library API, for people writing Sage library code that uses random numbers.

42.1 Command line oriented API

We'll start with the simplest usage: setting fixed random number seeds and showing that these lead to reproducible results.

```
sage: K.<x> = QQ[]
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: rgp = Gp()
sage: def gap_randstring(n):
....:     current_randstate().set_seed_gap()
....:     return gap(n).SCRRandomString()
sage: def rtest():
....:     current_randstate().set_seed_gp(rgp)
....:     return (ZZ.random_element(1000), RR.random_element(),
....:             K.random_element(), G.random_element(),
....:             gap_randstring(5),
....:             rgp.random(), ntl.ZZ_random(99999),
....:             random())
```

The above test shows the results of six different random number generators, in three different processes. The random elements from `ZZ`, `RR`, and `K` all derive from a single GMP-based random number generator. The random element from `G` comes from a GAP subprocess. The random "string" (5-element binary list) is also from a GAP subprocess, using the "classical" GAP random generator. The random number from `rgp` is from a Pari/gp subprocess. NTL's `ZZ_random` uses a separate NTL random number generator in the main Sage process. And `random()` is from a Python `random.Random` object.

Here we see that setting the random number seed really does make the results of these random number generators reproducible.

```
sage: set_random_seed(0)
sage: rtest()
(303, -0.266166246380421, 1/2*x^2 - 1/95*x - 1/2, (1,3,2), [ 0, 0, 0, 0, 1 ], 265625921, 8045, 0.9661
sage: set_random_seed(1)
sage: rtest()
(978, 0.0557699430711638, -3*x^2 - 1/12, (1,3,2), [ 0, 1, 1, 0, 0 ], 807447831, 60359, 0.833507765419
sage: set_random_seed(2)
sage: rtest()
(207, -0.0141049486533456, 4*x^2 + 1/2, (1,3,2), [ 0, 0, 1, 0, 1 ], 1642898426, 27695, 0.199825651172
sage: set_random_seed(0)
sage: rtest()
(303, -0.266166246380421, 1/2*x^2 - 1/95*x - 1/2, (1,3,2), [ 0, 0, 0, 0, 1 ], 265625921, 8045, 0.9661
sage: set_random_seed(1)
sage: rtest()
(978, 0.0557699430711638, -3*x^2 - 1/12, (1,3,2), [ 0, 1, 1, 0, 0 ], 807447831, 60359, 0.833507765419
sage: set_random_seed(2)
sage: rtest()
(207, -0.0141049486533456, 4*x^2 + 1/2, (1,3,2), [ 0, 0, 1, 0, 1 ], 1642898426, 27695, 0.199825651172
```

Once we've set the random number seed, we can check what seed was used. (This is not the current random number state; it does not change when random numbers are generated.)

```
sage: set_random_seed(12345)
sage: initial_seed()
12345L
sage: rtest()
(720, -0.612180244315804, x^2 - x, (2,3), [ 1, 0, 0, 0, 0 ], 1911581957, 14005, 0.9205331599518184)
sage: initial_seed()
12345L
```

If `set_random_seed()` is called with no arguments, then a new seed is automatically selected. On operating systems that support it, the new seed comes from `os.urandom()`; this is intended to be a truly random (not pseudo-random), cryptographically secure number. (Whether it is actually cryptographically secure depends on operating system details that are outside the control of Sage.)

If `os.urandom()` is not supported, then the new seed comes from the current time, which is definitely not cryptographically secure.

```
sage: set_random_seed()
sage: r = rtest()
sage: r          # random
(909, -0.407373370020575, 6/7*x^2 + 1, (1,2,3)(4,5), 985329107, 21461, 0.30047071049504859)
```

After setting a new random number seed with `set_random_seed()`, we can use `initial_seed()` to see what seed was automatically selected, and call `set_random_seed()` to restart the same random number sequence.

```
sage: s = initial_seed()
sage: s          # random
336237747258024892084418842839280045662L
sage: set_random_seed(s)
sage: r2 = rtest()
sage: r == r2
True
```

Whenever Sage starts, `set_random_seed()` is called just before command line interaction starts; so every Sage run starts with a different random number seed. This seed can be recovered with `initial_seed()` (as long as the user has not set a different seed with `set_random_seed()`), so that the results of this run can be reproduced in another run; or this automatically selected seed can be overridden with, for instance, `set_random_seed(0)`.

We can demonstrate this startup behavior by running a new instance of Sage as a subprocess.

```
sage: subsage = Sage()
sage: s = ZZ(subsage('initial_seed()'))
sage: r = ZZ(subsage('ZZ.random_element(2^200)'))
sage: s          # random
161165040149656168853863459174502758403
sage: r          # random
1273828861620427462924151488498075119241254209468761367941442
sage: set_random_seed(s)
sage: r == ZZ.random_element(2^200)
True
```

Note that wrappers of all the random number generation methods from Python's `random` module are available at the Sage command line, and these wrappers are properly affected by `set_random_seed()`.

```
sage: set_random_seed(0)
sage: random(), getrandbits(20), uniform(5.0, 10.0), normalvariate(0, 1)
(0.111439293741037, 539332L, 8.26785106378383, 1.3893337539828183)
sage: set_random_seed(1)
sage: random(), getrandbits(20), uniform(5.0, 10.0), normalvariate(0, 1)
(0.8294022851874259, 624859L, 5.77894484361117, -0.4201366826308758)
sage: set_random_seed(0)
sage: random(), getrandbits(20), uniform(5.0, 10.0), normalvariate(0, 1)
(0.111439293741037, 539332L, 8.26785106378383, 1.3893337539828183)
```

That pretty much covers what you need to know for command-line use of this module. Now let's move to what authors of Sage library code need to know about the module.

42.2 Library API

First, we'll cover doctesting. Every docstring now has an implicit `set_random_seed(0)` prepended. Any uses of `# random` that are based on random numbers under the control of this module should be removed, and the reproducible answers inserted instead.

This practice has two potential drawbacks. First, it increases the work of maintaining doctests. For instance, in a long docstring that has many doctests that depend on random numbers, a change near the beginning (for instance, adding a new doctest) may invalidate all later doctests in the docstring. To reduce this downside, you may add calls to `set_random_seed(0)` throughout the docstring (in the extreme case, before every doctest).

Second, the `# random` in the doctest served as a signal to the reader of the docstring that the result was unpredictable and that it would not be surprising to get a different result when trying out the examples in the doctest. If a doctest specifically refers to `ZZ.random_element()` (for instance), this is presumably enough of a signal to render this function of `# random` unnecessary. However, some doctests are not obviously (from the name) random, but do depend on random numbers internally, such as the `composition_series` method of a `PermutationGroup`. In these cases, the convention is to insert the following text at the beginning of the `EXAMPLES` section.

```
These computations use pseudo-random numbers, so we set the
seed for reproducible testing.

    sage: set_random_seed(0)
```

Note that this call to `set_random_seed(0)` is redundant, since `set_random_seed(0)` is automatically inserted at the beginning of every docstring. However, it makes the example reproducible for somebody who just types the lines from the doctest and doesn't know about the automatic `set_random_seed(0)`.

Next, let's cover setting the random seed from library code. The first rule is that library code should never call `set_random_seed()`. This function is only for command-line use. Instead, if the library code wants to use a different random seed, it should use `with seed(s):`. This will use the new seed within the scope of the `with` statement, but will revert to the previous seed once the `with` statement is completed. (Or the library can use `with seed():` to get a seed automatically selected using `os.urandom()` or the current time, in the same way as described for `set_random_seed()` above.)

Ideally, using `with seed(s):` should not affect the outer random number sequence at all; we will call this property "isolation." We achieve isolation for most, but not all, of the random number generators in Sage (we fail for generators, such as NTL, that do not provide an API to retrieve the current random number state).

We'll demonstrate isolation. First, we show the sequence of random numbers that you get without intervening `with seed`.

```
sage: set_random_seed(0)
sage: r1 = rtest(); r1
(303, -0.266166246380421, 1/2*x^2 - 1/95*x - 1/2, (1,3,2), [ 0, 0, 0, 0, 1 ], 265625921, 8045, 0.9661
sage: r2 = rtest(); r2
(105, 0.642309615982449, -x^2 - x - 6, (1,2,3), [ 1, 0, 0, 1, 1 ], 53231108, 1271, 0.0017671550773822
```

We get slightly different results with an intervening `with seed`.

```
sage: set_random_seed(0)
sage: r1 == rtest()
True
sage: with seed(1): rtest()
(978, 0.0557699430711638, -3*x^2 - 1/12, (1,3,2), [ 0, 1, 1, 0, 0 ], 807447831, 60359, 0.833507765419
sage: r2m = rtest(); r2m
(105, 0.642309615982449, -x^2 - x - 6, (1,2,3), [ 1, 0, 0, 1, 1 ], 53231108, 19769, 0.001767155077382
sage: r2m == r2
False
```

We can see that `r2` and `r2m` are the same except for the call to `ntl.ZZ_random()`, which produces different results with and without the `with seed`.

However, we do still get a partial form of isolation, even in this case, as we see in this example:

```
sage: set_random_seed(0)
sage: r1 == rtest()
True
sage: with seed(1):
....:     rtest()
....:     rtest()
(978, 0.0557699430711638, -3*x^2 - 1/12, (1,3,2), [ 0, 1, 1, 0, 0 ], 807447831, 60359, 0.833507765419
(138, -0.0404945051288503, 2*x - 24, (2,3), [ 1, 1, 1, 0, 1 ], 1010791326, 10234, 0.0033332230808060
sage: r2m == rtest()
True
```

The NTL results after the `with seed` don't depend on how many NTL random numbers were generated inside the `with seed`.

> sage: set_random_seed(0) sage: r1 == rtest() True sage: with seed(1): ....: rtest() (978, 0.0557699430711638, -3*x^2 - 1/12, (1,3,2), [0, 1, 1, 0, 0], 807447831, 60359, 0.8335077654199736) sage: r2m == rtest() True

(In general, the above code is not exactly equivalent to the `with` statement, because if an exception happens in the body, the real `with` statement will pass the exception information as parameters to the `__exit__` method. However, our `__exit__` method ignores the exception information anyway, so the above is equivalent in our case.)

42.3 Generating random numbers in library code

Now we come to the last part of the documentation: actually generating random numbers in library code. First, the easy case. If you generate random numbers only by calling other Sage library code (such as `random_element` methods on parents), you don't need to do anything special; the other code presumably already interacts with this module correctly.

Otherwise, it depends on what random number generator you want to use.

- `gmp_randstate_t` – If you want to use some random number generator that takes a `gmp_randstate_t` (like `mpz_urandomm` or `mpfr_urandomb`), then use code like the following:

  ```
  from sage.misc.randstate cimport randstate, current_randstate
  ...

  cdef randstate rstate = current_randstate()
  ```

  Then a `gmp_randstate_t` is available as `rstate.gmp_state`.

  Fetch the current `randstate` with `current_randstate()` in every function that wants to use it; don't cache it globally or in a class. (Such caching would break `set_random_seed`).

- `Python` – If you want to use the random number generators from the `random` module, you have two choices. The slightly easier choice is to import functions from `sage.misc.prandom`; for instance, you can simply replace `from random import randrange` with `from sage.misc.prandom import randrange`. However, this is slightly less efficient, because the wrappers in `sage.misc.prandom` look up the current `randstate` on each call. If you're generating many random numbers in a row, it's faster to instead do

  ```
  from sage.misc.randstate import current_randstate ...

  randrange = current_randstate().python_random().randrange
  ```

  Fetch the current `randstate` with `current_randstate()` in every function that wants to use it; don't cache the `randstate`, the `Random` object returned by `python_random`, or the bound methods on that `Random` object globally or in a class. (Such caching would break `set_random_seed`).

- `GAP` – If you are calling code in GAP that uses random numbers, call `set_seed_gap` at the beginning of your function, like this:

  ```
  from sage.misc.randstate import current_randstate
  ...

  current_randstate().set_seed_gap()
  ```

  Fetch the current `randstate` with `current_randstate()` in every function that wants to use it; don't cache it globally or in a class. (Such caching would break `set_random_seed`).

- `Pari` – If you are calling code in the Pari library that uses random numbers, call `set_seed_pari` at the beginning of your function, like this:

  ```
  from sage.misc.randstate import current_randstate
  ...

  current_randstate().set_seed_pari()
  ```

  Fetch the current `randstate` with `current_randstate()` in every function that wants to use it; don't cache it globally or in a class. (Such caching would break `set_random_seed`).

- `Pari/gp` – If you are calling code in a Pari/gp subprocess that uses random numbers, call `set_seed_gp` at the beginning of your function, like this:

  ```
  from sage.misc.randstate import current_randstate
  ...

  current_randstate().set_seed_gp()
  ```

  This will set the seed in the gp process in `sage.interfaces.gp.gp`. If you have a different gp process, say in the variable `my_gp`, then call `set_seed_gp(my_gp)` instead.

  Fetch the current `randstate` with `current_randstate()` in every function that wants to use it; don't cache it globally or in a class. (Such caching would break `set_random_seed`).

- `NTL` – If you are calling code in the NTL library that uses random numbers, call `set_seed_ntl` at the beginning of your function, like this:

  ```
  from sage.misc.randstate import current_randstate ...

  current_randstate().set_seed_ntl(False)
  ```

  Fetch the current `randstate` with `current_randstate()` in every function that wants to use it; don't cache it globally or in a class. (Such caching would break `set_random_seed`).

- `libc` – If you are writing code that calls the libc function `random()`: don't! The `random()` function does not give reproducible results across different operating systems, so we can't make portable doctests for the results. Instead, do:

  ```
  from sage.misc.randstate cimport random
  ```

  The `random()` function in `sage.misc.randstate` gives a 31-bit random number, but it uses the `gmp_randstate_t` in the current `randstate`, so it is portable. (This range was chosen for two reasons: it matches the range of random() on 32-bit and 64-bit Linux, although not Solaris; and it's the largest range of nonnegative numbers that fits in a 32-bit signed integer.)

  However, you may still need to set the libc random number state; for instance, if you are wrapping a library that uses `random()` internally and you don't want to change the library. In that case, call `set_seed_libc` at the beginning of your function, like this:

  ```
  from sage.misc.randstate import current_randstate
  ...

  current_randstate().set_seed_libc(False)
  ```

  Fetch the current `randstate` with `current_randstate()` in every function that wants to use it; don't cache it globally or in a class. (Such caching would break `set_random_seed`).

42.4 Classes and methods

`sage.misc.randstate.`**`benchmark_libc`**`()`

This function was used to test whether moving from libc to GMP's Mersenne Twister for random numbers would be a significant slowdown.

EXAMPLES:

```
sage: from sage.misc.randstate import benchmark_libc, benchmark_mt
sage: timeit('benchmark_libc()')  # random
125 loops, best of 3: 1.95 ms per loop
```

```
sage: timeit('benchmark_mt()')     # random
125 loops, best of 3: 2.12 ms per loop
```

`sage.misc.randstate.`**`benchmark_mt`**`()`

This function was used to test whether moving from libc to GMP's Mersenne Twister for random numbers would be a significant slowdown.

EXAMPLES:

```
sage: from sage.misc.randstate import benchmark_libc, benchmark_mt
sage: timeit('benchmark_libc()')   # random
125 loops, best of 3: 1.95 ms per loop
sage: timeit('benchmark_mt()')     # random
125 loops, best of 3: 2.11 ms per loop
```

`sage.misc.randstate.`**`current_randstate`**`()`

Return the current random number state.

EXAMPLES:

```
sage: current_randstate()
<sage.misc.randstate.randstate object at 0x...>
sage: current_randstate().python_random().random()
0.111439293741037
```

`sage.misc.randstate.`**`initial_seed`**`()`

Returns the initial seed used to create the current `randstate`.

EXAMPLES:

```
sage: set_random_seed(42)
sage: initial_seed()
42L
```

If you set a random seed (by failing to specify the seed), this is how you retrieve the seed actually chosen by Sage. This can also be used to retrieve the seed chosen for a new Sage run (if the user has not used `set_random_seed()`).

```
sage: set_random_seed()
sage: initial_seed()          # random
121030915255244661507561642968348336774L
```

`sage.misc.randstate.`**`random`**`()`

Returns a 31-bit random number. Intended as a drop-in replacement for the libc `random()` function.

EXAMPLES:

```
sage: set_random_seed(31)
sage: from sage.misc.randstate import random
sage: random()
32990711
```

class `sage.misc.randstate.`**`randstate`**

Bases: `object`

The `randstate` class. This class keeps track of random number states and seeds. Type `sage.misc.randstate?` for much more information on random numbers in Sage.

`ZZ_seed``()`

When called on the current `randstate`, returns a 128-bit `Integer` suitable for seeding another random number generator.

EXAMPLES:

```
sage: set_random_seed(1414)
sage: current_randstate().ZZ_seed()
48314508034782595865062786044921182484
```

c_rand_double()

Returns a random floating-point number between 0 and 1.

EXAMPLES:

```
sage: set_random_seed(2718281828)
sage: current_randstate().c_rand_double()
0.22437207488974298
```

c_random()

Returns a 31-bit random number. Intended for internal use only; instead of calling `current_randstate().c_random()`, it is equivalent (but probably faster) to call the `random` method of this `randstate` class.

EXAMPLES:

```
sage: set_random_seed(1207)
sage: current_randstate().c_random()
2008037228
```

We verify the equivalence mentioned above.

```
sage: from sage.misc.randstate import random
sage: set_random_seed(1207)
sage: random()
2008037228
```

long_seed()

When called on the current `randstate`, returns a 128-bit Python long suitable for seeding another random number generator.

EXAMPLES:

```
sage: set_random_seed(1618)
sage: current_randstate().long_seed()
256056279774514099508607350947089272595L
```

python_random()

Return a `random.Random` object. The first time it is called on a given `randstate`, a new `random.Random` is created (seeded from the *current* `randstate`); the same object is returned on subsequent calls.

It is expected that `python_random` will only be called on the current `randstate`.

EXAMPLES:

```
sage: set_random_seed(5)
sage: rnd = current_randstate().python_random()
sage: rnd.random()
0.013558022446944151
sage: rnd.randrange(1000)
544
```

seed()

Return the initial seed of a `randstate` object. (This is not the current state; it does not change when you get random numbers.)

EXAMPLES:

```
sage: from sage.misc.randstate import randstate
sage: r = randstate(314159)
sage: r.seed()
314159L
sage: r.python_random().random()
0.111439293741037
sage: r.seed()
314159L
```

set_seed_gap()

Checks to see if `self` was the most recent `randstate` to seed the GAP random number generator. If not, seeds the generator.

EXAMPLES:

```
sage: set_random_seed(99900000999)
sage: current_randstate().set_seed_gap()
sage: gap.Random(1, 10^50)
1496738263332555434474532297768680634540939580077
sage: gap(35).SCRRandomString()
[ 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0,
  0, 0, 1, 0, 0, 1, 1, 0, 0, 1 ]
```

set_seed_gp(*gp=None*)

Checks to see if `self` was the most recent `randstate` to seed the random number generator in the given instance of gp. (If no instance is given, uses the one in `gp`.) If not, seeds the generator.

EXAMPLES:

```
sage: set_random_seed(987654321)
sage: current_randstate().set_seed_gp()
sage: gp.random()
23289294
```

set_seed_libc(*force*)

Checks to see if `self` was the most recent `randstate` to seed the libc random number generator. If not, seeds the libc random number generator. (Do not use the libc random number generator if you have a choice; its randomness is poor, and the random number sequences it produces are not portable across operating systems.)

If the argument `force` is `True`, seeds the generator unconditionally.

EXAMPLES:

```
sage: from sage.misc.randstate import _doctest_libc_random
sage: set_random_seed(0xBAD)
sage: current_randstate().set_seed_libc(False)
sage: _doctest_libc_random()    # random
1070075918
```

set_seed_ntl(*force*)

Checks to see if `self` was the most recent `randstate` to seed the NTL random number generator. If not, seeds the generator. If the argument `force` is `True`, seeds the generator unconditionally.

EXAMPLES:

```
sage: set_random_seed(2008)
```

This call is actually redundant; `ntl.ZZ_random()` will seed the generator itself. However, we put the call in to make the coverage tester happy.

```
sage: current_randstate().set_seed_ntl(False)
sage: ntl.ZZ_random(10^40)
9121810031060285085432621721962973171231
```

set_seed_pari()

Checks to see if `self` was the most recent `randstate` to seed the Pari random number generator. If not, seeds the generator.

Note: Since pari 2.4.3, pari's random number generator has changed a lot. the seed output by getrand() is now a vector of integers.

EXAMPLES:

```
sage: set_random_seed(5551212)
sage: current_randstate().set_seed_pari()
sage: pari.getrand().type()
't_INT'
```

sage.misc.randstate.**seed**

alias of `randstate`

sage.misc.randstate.**set_random_seed**(*seed=None*)

Set the current random number seed from the given `seed` (which must be coercible to a Python long).

If no seed is given, then a seed is automatically selected using `os.urandom()` if it is available, or the current time otherwise.

Type `sage.misc.randstate?` for much more information on random numbers in Sage.

This function is only intended for command line use. Never call this from library code; instead, use `with seed(s):`.

Note that setting the random number seed to 0 is much faster than using any other number.

EXAMPLES:

```
sage: set_random_seed(5)
sage: initial_seed()
5L
```

CHAPTER

FORTYTHREE

CYTHON – C-EXTENSIONS FOR PYTHON

AUTHORS:

- William Stein (2006-01-18): initial version
- William Stein (2007-07-28): update from sagex to cython
- Martin Albrecht & William Stein (2011-08): cfile & cargs

sage.misc.cython.**atlas**()

Returns the name of the ATLAS library to use. On Darwin or Cygwin, this is `'blas'`, and otherwise it is `'atlas'`.

EXAMPLES:

```
sage: sage.misc.cython.atlas() # random -- depends on OS
'atlas'
```

sage.misc.cython.**cblas**()

Return the name of the cblas library on this system. If the environment variable `$SAGE_CBLAS` is set, just return its value. If not, return `'cblas'` if `/usr/lib/libcblas.so` or `/usr/lib/libcblas.dylib` exists, return `'blas'` if `/usr/lib/libblas.dll.a` exists, and return `'gslcblas'` otherwise.

EXAMPLES:

```
sage: sage.misc.cython.cblas() # random -- depends on OS, etc.
'cblas'
```

sage.misc.cython.**compile_and_load**(*code*)

INPUT:

- `code` – string containing code that could be in a .pyx file that is attached or put in a %cython block in the notebook.

OUTPUT: a module, which results from compiling the given code and importing it

EXAMPLES:

```
sage: module = sage.misc.cython.compile_and_load("def f(int n):\n    return n*n")
sage: module.f(10)
100
```

sage.misc.cython.**cython**(*filename*, *verbose=False*, *compile_message=False*, *use_cache=False*, *create_local_c_file=False*, *annotate=True*, *sage_namespace=True*, *create_local_so_file=False*)

Compile a Cython file. This converts a Cython file to a C (or C++ file), and then compiles that. The .c file and the .so file are created in a temporary directory.

INPUTS:

- `filename` - the name of the file to be compiled. Should end with 'pyx'.
- `verbose` (bool, default False) - if True, print debugging information.
- `compile_message` (bool, default False) - if True, print `'Compiling <filename>...'` to the standard error.
- `use_cache` (bool, default False) - if True, check the temporary build directory to see if there is already a corresponding .so file. If so, and if the .so file is newer than the Cython file, don't recompile, just reuse the .so file.
- `create_local_c_file` (bool, default False) - if True, save a copy of the .c file in the current directory.
- `annotate` (bool, default True) - if True, create an html file which annotates the conversion from .pyx to .c. By default this is only created in the temporary directory, but if `create_local_c_file` is also True, then save a copy of the .html file in the current directory.
- `sage_namespace` (bool, default True) - if True, import `sage.all`.
- `create_local_so_file` (bool, default False) - if True, save a copy of the compiled .so file in the current directory.

TESTS:

Before trac ticket #12975, it would have beeen needed to write `#clang c++`, but upper case `C++` has resulted in an error:

```
sage: code = [
... "#clang C++",
... "#cinclude %s/include/singular %s/include/factory"%(SAGE_LOCAL, SAGE_LOCAL),
... "#clib m readline singular givaro ntl gmpxx gmp",
... "from sage.rings.polynomial.multi_polynomial_libsingular cimport MPolynomial_libsingular",
... "from sage.libs.singular.polynomial cimport singular_polynomial_pow",
... "def test(MPolynomial_libsingular p):",
... "    singular_polynomial_pow(&p._poly, p._poly, 2, p._parent_ring)"]
sage: cython(os.linesep.join(code))
```

The function `test` now manipulates internal C data of polynomials, squaring them:

```
sage: P.<x,y>=QQ[]
sage: test(x)
sage: x
x^2
```

`sage.misc.cython.`**`cython_create_local_so`**(*filename*)

Compile filename and make it available as a loadable shared object file.

INPUT:

- `filename` - string: a Cython (.spyx) file

OUTPUT: None

EFFECT: A compiled, python "importable" loadable shared object file is created.

Note: Shared object files are *not* reloadable. The intent is for imports in other scripts. A possible development cycle might go thus:

- Attach a .spyx file
- Interactively test and edit it to your satisfaction
- Use `cython_create_local_so` to create the shared object file
- Import the .so file in other scripts

EXAMPLES:

```
sage: curdir = os.path.abspath(os.curdir)
sage: dir = tmp_dir(); os.chdir(dir)
sage: f = open('hello.spyx', 'w')
sage: s = "def hello():\n  print 'hello'\n"
sage: f.write(s)
sage: f.close()
sage: cython_create_local_so('hello.spyx')
Compiling hello.spyx...
sage: sys.path.append('.')
sage: import hello
sage: hello.hello()
hello
sage: os.chdir(curdir)
```

AUTHORS:

- David Fu (2008-04-09): initial version

sage.misc.cython.**cython_lambda**(*vars*, *expr*, *verbose=False*, *compile_message=False*, *use_cache=False*)

Create a compiled function which evaluates `expr` assuming machine values for `vars`.

INPUT:

- `vars` - list of pairs (variable name, c-data type), where the variable names and data types are strings, OR a string such as `'double x, int y, int z'`
- `expr` - an expression involving the vars and constants; you can access objects defined in the current module scope `globals()` using `sage.object_name`.

> **Warning:** Accessing `globals()` doesn't actually work, see trac ticket #12446.

EXAMPLES:

We create a Lambda function in pure Python (using the r to make sure the 3.2 is viewed as a Python float):

```
sage: f = lambda x,y: x*x + y*y + x + y + 17r*x + 3.2r
```

We make the same Lambda function, but in a compiled form.

```
sage: g = cython_lambda('double x, double y', 'x*x + y*y + x + y + 17*x + 3.2')
sage: g(2,3)
55.2
sage: g(0,0)
3.2
```

The following should work but doesn't, see trac ticket #12446:

```
sage: a = 25
sage: f = cython_lambda('double x', 'sage.math.sin(x) + sage.a')
sage: f(10)  # known bug
24.455978889110629
sage: a = 50
sage: f(10)  # known bug
49.455978889110632
```

sage.misc.cython.**environ_parse**(*s*)

Given a string s, find each substring of the form `'$ABC'`. If the environment variable `$ABC` is set, replace

`'$ABC'` with its value and move on to the next such substring. If it is not set, stop parsing there.

EXAMPLES:

```
sage: from sage.misc.cython import environ_parse
sage: environ_parse('$SAGE_LOCAL') == SAGE_LOCAL
True
sage: environ_parse('$THIS_IS_NOT_DEFINED_ANYWHERE')
'$THIS_IS_NOT_DEFINED_ANYWHERE'
sage: os.environ['DEFINE_THIS'] = 'hello'
sage: environ_parse('$DEFINE_THIS/$THIS_IS_NOT_DEFINED_ANYWHERE/$DEFINE_THIS')
'hello/$THIS_IS_NOT_DEFINED_ANYWHERE/$DEFINE_THIS'
```

`sage.misc.cython.`**`import_test`**(*name*)

This is used by the testing infrastructure to test building Cython programs.

INPUT:

- `name` – string; name of a key to the TESTS dictionary above

OUTPUT: a module, which results from compiling the given code and importing it

EXAMPLES:

```
sage: module = sage.misc.cython.import_test("trac11680b")
sage: module.f(2,3,4)
9
```

`sage.misc.cython.`**`parse_keywords`**(*kwd, s*)

Given a keyword `kwd` and a string `s`, return a list of all arguments on the same line as that keyword in `s`, as well as a new copy of `s` in which each occurrence of `kwd` is in a comment. If a comment already occurs on the line containing `kwd`, no words after the # are added to the list.

EXAMPLES:

```
sage: sage.misc.cython.parse_keywords('clib', " clib foo bar baz\n #cinclude bar\n")
(['foo', 'bar', 'baz'], ' #clib foo bar baz\n #cinclude bar\n')

sage: sage.misc.cython.parse_keywords('clib', "# qux clib foo bar baz\n #cinclude bar\n")
(['foo', 'bar', 'baz'], '# qux clib foo bar baz\n #cinclude bar\n')
sage: sage.misc.cython.parse_keywords('clib', "# clib foo bar # baz\n #cinclude bar\n")
(['foo', 'bar'], '# clib foo bar # baz\n #cinclude bar\n')
```

`sage.misc.cython.`**`pyx_preparse`**(*s*)

Preparse a pyx file:

- include `cdefs.pxi`, `interrupt.pxi`, `stdsage.pxi`
- parse `clang` pragma (c or c++)
- parse `clib` pragma (additional libraries to link in)
- parse `cinclude` (additional include directories)
- parse `cfile` (additional files to be included)
- parse `cargs` (additional parameters passed to the compiler)

The pragmas:

- `clang` - may be either `'c'` or `'c++'` indicating whether a C or C++ compiler should be used
- `clib` - additional libraries to be linked in, the space separated list is split and passed to distutils.

- `cinclude` - additional directories to search for header files. The space separated list is split and passed to distutils.
- `cfile` - additional C or C++ files to be compiled. Also, `$SAGE_SRC` and `$SAGE_LOCAL` are expanded, but other environment variables are not.
- `cargs` - additional parameters passed to the compiler

OUTPUT: preamble, libs, includes, language, files, args

EXAMPLES:

```
sage: from sage.misc.cython import pyx_preparse
sage: pyx_preparse("")
('\ninclude "interrupt.pxi"  # ctrl-c interrupt block support\ninclude "stdsage.pxi"\n\ninclude
['mpfr',
'gmp',
'gmpxx',
'stdc++',
'pari',
'm',
'ec',
'gsl',
'...blas',
...,
'ntl',
'csage'],
['.../include/csage',
'.../include',
'.../include/python2.7',
'.../lib/python/site-packages/numpy/core/include',
'.../sage/ext',
'...',
'.../sage/gsl'],
'c',
[], ['-w', '-O2'])
sage: s, libs, inc, lang, f, args = pyx_preparse("# clang c++\n #clib foo\n # cinclude bar\n")
sage: lang
'c++'

sage: libs
['foo', 'mpfr',
'gmp', 'gmpxx',
'stdc++',
'pari',
'm',
'ec',
'gsl', '...blas', ...,
'ntl',
'csage']
sage: libs[1:] == sage.misc.cython.standard_libs
True

sage: inc
['bar',
'.../include/csage',
'.../include',
'.../include/python2.7',
'.../lib/python/site-packages/numpy/core/include',
'.../sage/ext',
'...',
```

```
'.../sage/gsl']

sage: s, libs, inc, lang, f, args = pyx_preparse("# cargs -O3 -ggdb\n")
sage: args
['-w', '-O2', '-O3', '-ggdb']
```

TESTS:

```
sage: module = sage.misc.cython.import_test("trac11680")  # long time (7s on sage.math, 2012)
sage: R.<x> = QQ[]
sage: module.evaluate_at_power_of_gen(x^3 + x - 7, 5)  # long time
x^15 + x^5 - 7
```

`sage.misc.cython.`**`sanitize`**(*f*)

Given a filename `f`, replace it by a filename that is a valid Python module name.

This means that the characters are all alphanumeric or _'s and doesn't begin with a numeral.

EXAMPLES:

```
sage: from sage.misc.cython import sanitize
sage: sanitize('abc')
'abc'
sage: sanitize('abc/def')
'abc_def'
sage: sanitize('123/def-hij/file.py')
'_123_def_hij_file_py'
```

`sage.misc.cython.`**`subtract_from_line_numbers`**(*s*, *n*)

Given a string `s` and an integer `n`, for any line of `s` which has the form `'text:NUM:text'` subtract `n` from NUM and return `'text:(NUM-n):text'`. Return other lines of `s` without change.

EXAMPLES:

```
sage: from sage.misc.cython import subtract_from_line_numbers
sage: subtract_from_line_numbers('hello:1234:hello', 3)
'hello:1231:hello\n'
sage: subtract_from_line_numbers('text:123\nhello:1234:', 3)
'text:123\nhello:1231:\n'
```

CHAPTER

FORTYFOUR

MESSAGE DELIVERY.

Various interfaces to messaging services. Currently:

- `pushover` - a platform for sending and receiving push notifications

is supported.

AUTHORS:

- Martin Albrecht (2012) - initial implementation

`sage.misc.messaging.`**`pushover`**(*message*, ***kwds*)

Send a push notification with `message` to `user` using https://pushover.net/.

Pushover is a platform for sending and receiving push notifications. On the server side, it provides an HTTP API for queueing messages to deliver to devices. On the device side, iOS and Android clients receive those push notifications, show them to the user, and store them for offline viewing.

An account on https://pushover.net is required and the Pushover app must be installed on your phone for this function to be able to deliver messages to you.

INPUT:

- `message` - your message
- `user` - the user key (not e-mail address) of your user (or you), viewable when logged into the Pushover dashboard. (default: `None`)
- `device` - your user's device identifier to send the message directly to that device, rather than all of the user's devices (default: `None`)
- `title` - your message's title, otherwise uses your app's name (default: `None`)
- `url` - a supplementary URL to show with your message (default: `None`)
- `url_title` - a title for your supplementary URL (default: `None`)
- `priority` - set to 1 to display as high-priority and bypass quiet hours, or -1 to always send as a quiet notification (default: `0`)
- `timestamp` - set to a unix timestamp to have your message show with a particular time, rather than now (default: `None`)
- `sound` - set to the name of one of the sounds supported by device clients to override the user's default sound choice (default: `None`)
- `token` - your application's API token (default: Sage's default App token)

EXAMPLE:

```
sage: sage.misc.messaging.pushover("Hi, how are you?", user="XXX") # not tested
```

To set default values populate `pushover_defaults`:

```
sage: sage.misc.messaging.pushover_defaults["user"] = "USER_TOKEN"
sage: sage.misc.messaging.pushover("Hi, how are you?") # not tested
```

Note: You may want to populate `sage.misc.messaging.pushover_defaults` with default values such as the default user in `$HOME/.sage/init.sage`.

CHAPTER

FORTYFIVE

DETERMINATION OF PROGRAMS FOR VIEWING WEB PAGES, ETC.

The function `default_viewer()` defines reasonable defaults for these programs. To use something else, use `viewer`. First import it:

```
sage: from sage.misc.viewer import viewer
```

On OS X, PDFs are opened by default using the 'open' command, which runs whatever has been designated as the PDF viewer in the OS. To change this to use 'Adobe Reader':

```
sage: viewer.pdf_viewer('open -a /Applications/Adobe\ Reader.app') # not tested
```

Similarly, you can set `viewer.browser(...)`, `viewer.dvi_viewer(...)`, and `viewer.png_viewer(...)`. You can make this change permanent by adding lines like these to your `SAGE_STARTUP_FILE` (which is `$HOME/.sage/init.sage` by default):

```
from sage.misc.viewer import viewer
viewer.pdf_viewer('open -a /Applications/Adobe\ Reader.app')
```

45.1 Functions and classes

class `sage.misc.viewer.`**`Viewer`**

Bases: `sage.structure.sage_object.SageObject`

Set defaults for various viewing applications: a web browser, a dvi viewer, a pdf viewer, and a png viewer.

EXAMPLES:

```
sage: from sage.misc.viewer import viewer
sage: old_browser = viewer.browser()  # indirect doctest
sage: viewer.browser('open -a /Applications/Firefox.app')
sage: viewer.browser()
'open -a /Applications/Firefox.app'
sage: viewer.browser(old_browser) # restore old value
```

`browser`(*app=None*)

Change the default browser. Return the current setting if arg is `None`, which is the default.

INPUTS:

- `app` – `None` or a string, the program to use

EXAMPLES:

```
sage: from sage.misc.viewer import viewer
sage: old_browser = viewer.browser()
sage: viewer.browser('open -a /Applications/Firefox.app') # indirect doctest
sage: viewer.browser()
'open -a /Applications/Firefox.app'
sage: viewer.browser(old_browser) # restore old value
```

dvi_viewer(*app=None*)

Change the default dvi viewer. Return the current setting if arg is `None`, which is the default.

INPUTS:

- app – `None` or a string, the program to use

EXAMPLES:

```
sage: from sage.misc.viewer import viewer
sage: old_dvi_app = viewer.dvi_viewer()
sage: viewer.dvi_viewer('/usr/bin/xdvi') # indirect doctest
sage: viewer.dvi_viewer()
'/usr/bin/xdvi'
sage: viewer.dvi_viewer(old_dvi_app) # restore old value
```

pdf_viewer(*app=None*)

Change the default pdf viewer. Return the current setting if arg is `None`, which is the default.

INPUTS:

- app – `None` or a string, the program to use

EXAMPLES:

```
sage: from sage.misc.viewer import viewer
sage: old_pdf_app = viewer.pdf_viewer()
sage: viewer.pdf_viewer('/usr/bin/pdfopen') # indirect doctest
sage: viewer.pdf_viewer()
'/usr/bin/pdfopen'
sage: viewer.pdf_viewer(old_pdf_app) # restore old value
```

png_viewer(*app=None*)

Change the default png viewer. Return the current setting if arg is `None`, which is the default.

INPUTS:

- app – `None` or a string, the program to use

EXAMPLES:

```
sage: from sage.misc.viewer import viewer
sage: old_png_app = viewer.png_viewer()
sage: viewer.png_viewer('display') # indirect doctest
sage: viewer.png_viewer()
'display'
sage: viewer.png_viewer(old_png_app) # restore old value
```

sage.misc.viewer.**browser**()

Return the program used to open a web page. By default, the program used depends on the platform and other factors, like settings of certain environment variables. To use a different program, call `viewer.browser('PROG')`, where 'PROG' is the desired program.

This will start with 'sage-native-execute', which sets the environment appropriately.

EXAMPLES:

```
sage: from sage.misc.viewer import browser
sage: browser() # random -- depends on OS, etc.
'sage-native-execute sage-open'
sage: browser().startswith('sage-native-execute')
True
```

`sage.misc.viewer.default_viewer(viewer=None)`

Set up default programs for opening web pages, PDFs, PNGs, and DVI files.

INPUT:

- `viewer`: `None` or a string: one of 'browser', 'pdf', 'png', 'dvi' – return the name of the corresponding program. `None` is treated the same as 'browser'.

EXAMPLES:

```
sage: from sage.misc.viewer import default_viewer
sage: default_viewer(None) # random -- depends on OS, etc.
'sage-open'
sage: default_viewer('pdf') # random -- depends on OS, etc.
'xdg-open'
sage: default_viewer('jpg')
Traceback (most recent call last):
...
ValueError: Unknown type of viewer: jpg.
```

`sage.misc.viewer.dvi_viewer()`

Return the program used to display a dvi file. By default, the program used depends on the platform and other factors, like settings of certain environment variables. To use a different program, call `viewer.dvi_viewer('PROG')`, where 'PROG' is the desired program.

This will start with 'sage-native-execute', which sets the environment appropriately.

EXAMPLES:

```
sage: from sage.misc.viewer import dvi_viewer
sage: dvi_viewer() # random -- depends on OS, etc.
'sage-native-execute sage-open'
sage: dvi_viewer().startswith('sage-native-execute')
True
```

`sage.misc.viewer.pdf_viewer()`

Return the program used to display a pdf file. By default, the program used depends on the platform and other factors, like settings of certain environment variables. To use a different program, call `viewer.pdf_viewer('PROG')`, where 'PROG' is the desired program.

This will start with 'sage-native-execute', which sets the environment appropriately.

EXAMPLES:

```
sage: from sage.misc.viewer import pdf_viewer, viewer
sage: old_pdf_app = viewer.pdf_viewer()
sage: viewer.pdf_viewer('acroread')
sage: pdf_viewer()
'sage-native-execute acroread'
sage: viewer.pdf_viewer('old_pdf_app')
```

`sage.misc.viewer.png_viewer()`

Return the program used to display a png file. By default, the program used depends on the platform and other factors, like settings of certain environment variables. To use a different program, call `viewer.png_viewer('PROG')`, where 'PROG' is the desired program.

This will start with 'sage-native-execute', which sets the environment appropriately.

EXAMPLES:

```
sage: from sage.misc.viewer import png_viewer
sage: png_viewer() # random -- depends on OS, etc.
'sage-native-execute xdg-open'
sage: png_viewer().startswith('sage-native-execute')
True
```

CHAPTER FORTYSIX

LOADING AND SAVING SESSIONS AND LISTING ALL VARIABLES

EXAMPLES:

We reset the current session, then define a rational number 2/3, and verify that it is listed as a newly defined variable:

```
sage: reset()
sage: w = 2/3; w
2/3
sage: show_identifiers()
['w']
```

We next save this session. We are using a file in SAGE_TMP. We do this *for testing* only — please do not do this, when you want to save your session permanently, since SAGE_TMP will be removed when leaving Sage!

```
sage: save_session(os.path.join(SAGE_TMP, 'session'))
```

This saves a dictionary with w as one of the keys:

```
sage: z = load(os.path.join(SAGE_TMP, 'session'))
sage: z.keys()
['w']
sage: z['w']
2/3
```

Next we reset the session, verify this, and load the session back.:

```
sage: reset()
sage: show_identifiers()
[]
sage: load_session(os.path.join(SAGE_TMP, 'session'))
```

Indeed w is now defined again.:

```
sage: show_identifiers()
['w']
sage: w
2/3
```

It is not needed to clean up the file created in the above code, since it resides in the directory SAGE_TMP.

AUTHOR:

- William Stein

sage.misc.session.**init**(*state=None*)

Initialize some dictionaries needed by the show_identifiers(), save_session(), and load_session() functions.

INPUT:

- `state` – a dictionary or `None`; if `None` the `locals()` of the caller is used.

EXAMPLES:

```
sage: reset()
sage: w = 10
sage: show_identifiers()
['w']
```

When we call `init()` below it reinitializes the internal table, so the `w` we just defined doesn't count as a new identifier:

```
sage: sage.misc.session.init()
sage: show_identifiers()
[]
```

`sage.misc.session.`**`load_session`**(*name='sage_session'*, *verbose=False*)

Load a saved session.

This merges in all variables from a previously saved session. It does not clear out the variables in the current sessions, unless they are overwritten. You can thus merge multiple sessions, and don't necessarily loose all your current work when you use this command.

Note: In the Sage notebook the session name is searched for both in the current working cell and the `DATA` directory.

EXAMPLES:

```
sage: a = 5
sage: f = lambda x: x^2
```

For testing, we use a temporary file, that will be removed as soon as Sage is left. Of course, for permanently saving your session, you should choose a permanent file.

```
sage: tmp_f = tmp_filename()
sage: save_session(tmp_f)
sage: del a; del f
sage: load_session(tmp_f)
sage: print a
5
```

Note that `f` does not come back, since it is a function, hence couldn't be saved:

```
sage: print f
Traceback (most recent call last):
...
NameError: name 'f' is not defined
```

`sage.misc.session.`**`save_session`**(*name='sage_session'*, *verbose=False*)

Save all variables that can be saved to the given filename. The variables will be saved to a dictionary, which can be loaded using `load(name)` or `load_session()`.

Note:

1. Function and anything else that can't be pickled is not saved. This failure is silent unless you set `verbose=True`.
2. In the Sage notebook the session is saved both to the current working cell and to the `DATA` directory.
3. One can still make sessions that can't be reloaded. E.g., define a class with:

```
class Foo: pass
```

and make an instance with:

```
f = Foo()
```

Then `save_session()` followed by `quit` and `load_session()` fails. I doubt there is any good way to deal with this. Fortunately, one can simply re-evaluate the code to define `Foo`, and suddenly `load_session()` works fine.

INPUT:

- `name` – string (default: 'sage_session') name of `sobj` to save the session to.
- `verbose` – bool (default: `False`) if `True`, print info about why certain variables can't be saved.

OUTPUT:

- Creates a file and returns silently.

EXAMPLES:

For testing, we use a temporary file that will be removed as soon as Sage is left. Of course, for permanently saving your session, you should choose a permanent file.

```
sage: a = 5
sage: tmp_f = tmp_filename()
sage: save_session(tmp_f)
sage: del a
sage: load_session(tmp_f)
sage: print a
5
```

We illustrate what happens when one of the variables is a function:

```
sage: f = lambda x : x^2
sage: save_session(tmp_f)
sage: save_session(tmp_f, verbose=True)
Saving...
Not saving f: f is a function, method, class or type
...
```

Something similar happens for cython-defined functions:

```
sage: g = cython_lambda('double x', 'x*x + 1.5')
sage: save_session(tmp_f, verbose=True)
Not saving g: g is a function, method, class or type
...
```

`sage.misc.session.`**`show_identifiers`**(*hidden=False*)

Returns a list of all variable names that have been defined during this session. By default, this returns only those identifiers that don't start with an underscore.

INPUT:

- `hidden` – bool (Default: `False`); If `True`, also return identifiers that start with an underscore.

OUTPUT:

A list of variable names

EXAMPLES:

We reset the state of all variables, and see that none are defined:

```
sage: reset()
sage: show_identifiers()
[]
```

We then define two variables, one which overwrites the default factor function; both are shown by `show_identifiers()`:

```
sage: a = 10
sage: factor = 20
sage: show_identifiers()
['a', 'factor']
```

To get the actual value of a variable from the list, use the `globals()` function.:

```
sage: globals()['factor']
20
```

By default `show_identifiers()` only returns variables that don't start with an underscore. There is an option hidden that allows one to list those as well:

```
sage: _hello = 10
sage: show_identifiers()
['a', 'factor']
sage: '_hello' in show_identifiers(hidden=True)
True
```

Many of the hidden variables are part of the IPython command history, at least in command line mode.:

```
sage: show_identifiers(hidden=True)        # random output
['__', '_i', '_6', '_4', '_3', '_1', '_ii', '__doc__', '__builtins__', '___', '_9', '__name__',
```

CHAPTER

FORTYSEVEN

DEFAULT SETTINGS

AUTHORS: William Stein and David Kohel

`sage.misc.defaults.`**`latex_variable_names`**`(`*n*, *name=None*`)`

x.__init__(...) initializes x; see help(type(x)) for signature

`sage.misc.defaults.`**`series_precision`**`()`

Return the Sage-wide precision for series (symbolic, power series, Laurent series).

EXAMPLES:

```
sage: series_precision()
20
```

`sage.misc.defaults.`**`set_default_variable_name`**`(`*name*, *separator=''*`)`

Change the default variable name and separator.

`sage.misc.defaults.`**`set_series_precision`**`(`*prec*`)`

Change the Sage-wide precision for series (symbolic, power series, Laurent series).

EXAMPLES:

```
sage: set_series_precision(5)
sage: series_precision()
5
sage: set_series_precision(20)
```

`sage.misc.defaults.`**`variable_names`**`(`*n*, *name=None*`)`

x.__init__(...) initializes x; see help(type(x)) for signature

CHAPTER

FORTYEIGHT

HANDLING SUPERSEDED FUNCTIONALITY

The main mechanism in Sage to deal with superseded functionality is to add a deprecation warning. This will be shown once, the first time that the deprecated function is called.

Note that all doctests in the following use the trac ticket number trac ticket #13109, which is where this mandatory argument to `deprecation()` was introduced.

48.1 Functions and classes

class `sage.misc.superseded.`**`DeprecatedFunctionAlias`**(*trac_number*, *func*, *module*)

Bases: `object`

A wrapper around methods or functions which automatically print the correct deprecation message. See `deprecated_function_alias()`.

AUTHORS:

- Florent Hivert (2009-11-23), with the help of Mike Hansen.
- Luca De Feo (2011-07-11), printing the full module path when different from old path

`sage.misc.superseded.`**`deprecated_callable_import`**(*trac_number*, *module_name*, *globs*, *locs*, *fromlist*, *message=None*)

Imports a list of callables into the namespace from which it is called. These callables however give a deprecation warning whenever they are called. This is primarily used from deprecating things from Sage's `all.py` files.

INPUT:

- `trac_number` – integer. The trac ticket number where the deprecation is introduced.
- `param module_name` – string or `None`. The name of the module from which to import the callables or `None`.
- `globs` – dictionary. The `globals()` from where this is being called.
- `locs` – dictionary. The `locals()` from where this is being called.
- `param fromlist:` – list of strings. The list the names of the callables to deprecate
- `message` –' string. Message to display when the deprecated functions are called.

Note: If `module_name` is `None`, then no importing will be done, and it will be assumed that the functions have already been imported and are present in `globs`

Warning: This should really only be used for functions.

EXAMPLES:

```
sage: from sage.misc.superseded import deprecated_callable_import
sage: is_prime(3)
True
sage: message = "Using %(name)s from here is deprecated."
sage: deprecated_callable_import(13109, None, globals(), locals(), ['is_prime'], message)
sage: is_prime(3)
doctest:...: DeprecationWarning:
Using is_prime from here is deprecated.
See http://trac.sagemath.org/13109 for details.
True
sage: del is_prime
sage: deprecated_callable_import(13109, 'sage.rings.arith', globals(), locals(), ['is_prime'])
sage: is_prime(3)
doctest:...: DeprecationWarning:
Using is_prime from here is deprecated.  If you need to use it, please import it directly from s
See http://trac.sagemath.org/13109 for details.
True
```

sage.misc.superseded.**deprecated_function_alias**(*trac_number*, *func*)

Create an aliased version of a function or a method which raise a deprecation warning message.

If f is a function or a method, write `g = deprecated_function_alias(trac_number, f)` to make a deprecated aliased version of f.

INPUT:

- `trac_number` – integer. The trac ticket number where the deprecation is introduced.
- `func` – the function or method to be aliased

EXAMPLES:

```
sage: from sage.misc.superseded import deprecated_function_alias
sage: g = deprecated_function_alias(13109, number_of_partitions)
sage: g(5)
doctest:...: DeprecationWarning: g is deprecated. Please use sage.combinat.partition.number_of_p
See http://trac.sagemath.org/13109 for details.
7
```

This also works for methods:

```
sage: class cls(object):
....:    def new_meth(self): return 42
....:    old_meth = deprecated_function_alias(13109, new_meth)
sage: cls().old_meth()
doctest:...: DeprecationWarning: old_meth is deprecated. Please use new_meth instead.
See http://trac.sagemath.org/13109 for details.
42
```

trac ticket #11585:

```
sage: def a(): pass
sage: b = deprecated_function_alias(13109, a)
sage: b()
doctest:...: DeprecationWarning: b is deprecated. Please use a instead.
See http://trac.sagemath.org/13109 for details.
```

AUTHORS:

- Florent Hivert (2009-11-23), with the help of Mike Hansen.

- Luca De Feo (2011-07-11), printing the full module path when different from old path

`sage.misc.superseded.`**`deprecation`**(*trac_number*, *message*)

Issue a deprecation warning.

INPUT:

- `trac_number` – integer. The trac ticket number where the deprecation is introduced.
- `message` – string. an explanation why things are deprecated and by what it should be replaced.

EXAMPLES:

```
sage: def foo():
....:  sage.misc.superseded.deprecation(13109, 'the function foo is replaced by bar')
sage: foo()
doctest:...: DeprecationWarning: the function foo is replaced by bar
See http://trac.sagemath.org/13109 for details.
```

CHAPTER

FORTYNINE

PROFILING AND PERFORMANCE TESTING

49.1 Accurate timing information for Sage commands

This is an implementation of nice timeit functionality, like the `%timeit` magic command in IPython. To use it, use the `timeit` command. This command then calls `sage_timeit()`, which you can find below.

EXAMPLES:

```
sage: timeit('1+1')     # random output
625 loops, best of 3: 314 ns per loop
```

AUTHOR:

– William Stein, based on code by Fernando Perez included in IPython

class `sage.misc.sage_timeit.`**`SageTimeitResult`**(*stats*, *series=None*)

Bases: `object`

Represent the statistics of a timeit() command.

Prints as a string so that it can be easily returned to a user.

INPUT:

- `stats` – tuple of length 5 containing the following information:
  - integer, number of loops
  - integer, repeat number
  - Python integer, number of digits to print
  - number, best timing result
  - str, time unit

EXAMPLES:

```
sage: from sage.misc.sage_timeit import SageTimeitResult
sage: SageTimeitResult( (3, 5, int(8), pi, 'ms') )
3 loops, best of 5: 3.1415927 ms per loop

sage: units = ["s", "ms", "\xc2\xb5s", "ns"]
sage: scaling = [1, 1e3, 1e6, 1e9]
sage: number = 7
sage: repeat = 13
sage: precision = int(5)
sage: best = pi / 10 ^ 9
sage: order = 3
```

```
sage: stats = (number, repeat, precision, best * scaling[order], units[order])
sage: SageTimeitResult(stats)
7 loops, best of 13: 3.1416 ns per loop
```

If the third argument is not a Python integer, a `TypeError` is raised:

```
sage: SageTimeitResult( (1, 2, 3, 4, 's') )
<repr(<sage.misc.sage_timeit.SageTimeitResult at 0x...>) failed: TypeError: * wants int>
```

`sage.misc.sage_timeit.`**`sage_timeit`**(*stmt*, *globals_dict=None*, *preparse=None*, *number=0*, *repeat=3*, *precision=3*, *seconds=False*)

Accurately measure the wall time required to execute `stmt`.

INPUT:

- `stmt` – a text string.
- `globals_dict` – a dictionary or `None` (default). Evaluate `stmt` in the context of the globals dictionary. If not set, the current `globals()` dictionary is used.
- `preparse` – (default: use globals preparser default) if `True` preparse `stmt` using the Sage preparser.
- `number` – integer, (optional, default: 0), number of loops.
- `repeat` – integer, (optional, default: 3), number of repetition.
- `precision` – integer, (optional, default: 3), precision of output time.
- `seconds` – boolean (default: `False`). Whether to just return time in seconds.

OUTPUT:

An instance of `SageTimeitResult` unless the optional parameter `seconds=True` is passed. In that case, the elapsed time in seconds is returned as a floating-point number.

EXAMPLES:

```
sage: from sage.misc.sage_timeit import sage_timeit
sage: sage_timeit('3^100000', globals(), preparse=True, number=50)      # random output
'50 loops, best of 3: 1.97 ms per loop'
sage: sage_timeit('3^100000', globals(), preparse=False, number=50)     # random output
'50 loops, best of 3: 67.1 ns per loop'
sage: a = 10
sage: sage_timeit('a^2', globals(), number=50)                            # random output
'50 loops, best of 3: 4.26 us per loop'
```

If you only want to see the timing and not have access to additional information, just use the `timeit` object:

```
sage: timeit('10^2', number=50)
50 loops, best of 3: ... per loop
```

Using sage_timeit gives you more information though:

```
sage: s = sage_timeit('10^2', globals(), repeat=1000)
sage: len(s.series)
1000
sage: mean(s.series)   # random output
3.1298141479492283e-07
sage: min(s.series)    # random output
2.9258728027343752e-07
sage: t = stats.TimeSeries(s.series)
sage: t.scale(10^6).plot_histogram(bins=20,figsize=[12,6], ymax=2)
Graphics object consisting of 20 graphics primitives
```

The input expression can contain newlines (but doctests cannot, so we use `os.linesep` here):

```
sage: from sage.misc.sage_timeit import sage_timeit
sage: from os import linesep as CR
sage: # sage_timeit(r'a = 2\nb=131\nfactor(a^b-1)')
sage: sage_timeit('a = 2' + CR + 'b=131' + CR + 'factor(a^b-1)',
...                 globals(), number=10)
10 loops, best of 3: ... per loop
```

Test to make sure that `timeit` behaves well with output:

```
sage: timeit("print 'Hi'", number=50)
50 loops, best of 3: ... per loop
```

If you want a machine-readable output, use the `seconds=True` option:

```
sage: timeit("print 'Hi'", seconds=True)   # random output
1.42555236816e-06
sage: t = timeit("print 'Hi'", seconds=True)
sage: t     #r random output
3.6010742187499999e-07
```

TESTS:

Make sure that garbage collection is re-enabled after an exception occurs in timeit:

```
sage: def f(): raise ValueError
sage: import gc
sage: gc.isenabled()
True
sage: timeit("f()")
Traceback (most recent call last):
...
ValueError
sage: gc.isenabled()
True
```

49.2 The `timeit` command

This uses the function `sage_timeit()`.

class `sage.misc.sage_timeit_class.`**`SageTimeit`**

Time execution of a command or block of commands.

Displays the best WALL TIME for execution of the given code. This is based on the Python timeit module, which avoids a number of common traps for measuring execution times. It is also based on IPython's %timeit command.

TYPICAL INPUT FORMAT:

```
timeit(statement, preparse=None, number=0, repeat=3, precision=3)
```

EXAMPLES:

```
sage: timeit('2^10000')
625 loops, best of 3: ... per loop
```

We illustrate some options:

```
sage: timeit('2+2',precision=2,number=20,repeat=5)
20 loops, best of 5: ... per loop
```

The preparser is on by default (if it is on), but the preparse option allows us to override it:

```
sage: timeit('2^10000', preparse=False, number=50)
50 loops, best of 3: ... per loop
```

The input can contain newlines:

```
sage: timeit("a = 2\nb=131\nfactor(a^b-1)", number=25)
25 loops, best of 3: ... per loop
```

See also:

`runsnake()`

`eval` (*code*, *globs=None*, *locals=None*, ***kwds*)

This eval function is called when doing %timit in the notebook.

INPUT:

- `code` – string of code to evaluate; may contain newlines.
- `globs` – global variables; if not given, uses module scope globals.
- `locals` – ignored completely.
- `kwds` – passed onto sage_timeit. Common options are `preparse`, `number`, `repeat`, `precision`. See `sage_timeit()` for details.

OUTPUT: string – timing information as a string

EXAMPLES:

```
sage: timeit.eval("2+2")                         # random output
'625 loops, best of 3: 1.47 us per loop'
```

We emphasize that timeit times WALL TIME. This is good in the context of Sage where commands often call out to other subprocesses that don't appear in CPU time.

```
sage: timeit('sleep(0.5)', number=3)   # long time (5s on sage.math, 2012)
3 loops, best of 3: ... ms per loop
```

49.3 Simple profiling tool

AUTHORS:

- David Harvey (August 2006)
- Martin Albrecht

class `sage.misc.profiler.Profiler` (*systems=[ ]*, *verbose=False*)

Keeps track of CPU time used between a series of user-defined checkpoints.

It's probably not a good idea to use this class in an inner loop :-)

EXAMPLE:

```
sage: def f():                          # not tested
...       p = Profiler()                # not tested
```

Calling p(message) creates a checkpoint:

```
sage: p("try factoring 15")             # not tested
```

Do something time-consuming:

```
sage: x = factor(15)                    # not tested
```

You can create a checkpoints without a string; Profiler will use the source code instead:

```
sage: p()                               # not tested
sage: y = factor(25)                    # not tested
sage: p("last step")                    # not tested
sage: z = factor(35)                    # not tested
sage: p()                               # not tested
```

This will give a nice list of timings between checkpoints:

```
sage: print p                           # not tested
```

Let's try it out:

```
sage: f()                               # not tested
    3.020s -- try factoring 15
   15.240s -- line 17: y = factor(25)
 5000.190s -- last step
```

See also:

runsnake()

Todo

- Add Pyrex source code inspection (I assume it doesn't currently do this)
- Add ability to sort output by time
- Add option to constructor to print timing immediately when checkpoint is reached
- Migrate to Pyrex?
- Add ability to return timings in a more machine-friendly format

---

AUTHOR:

- David Harvey (August 2006)

clear()

x.__init__(...) initializes x; see help(type(x)) for signature

print_last()

Prints the last profiler step

49.4 C Function Profiler Using Google Perftools

Note that the profiler samples 100x per second by default. In particular, you cannot profile anything shorter than 10ms. You can adjust the rate with the `CPUPROFILE_FREQUENCY` evironment variable if you want to change it.

EXAMPLES:

```
sage: from sage.misc.gperftools import Profiler, run_100ms
sage: prof = Profiler()
sage: prof.start()        # optional - gperftools
sage: run_100ms()
sage: prof.stop()         # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
```

REFERENCE:

Uses the Google performance analysis tools. Note that they are not included in Sage, you have to install them yourself on your system.

AUTHORS:

- Volker Braun (2014-03-31): initial version

class `sage.misc.gperftools.`**`Profiler`**(*filename=None*)

Bases: `sage.structure.sage_object.SageObject`

Interface to the gperftools profiler

INPUT:

- `filename` – string or `None` (default). The file name to log to. By default, a new temporary file is created.

EXAMPLES:

```
sage: from sage.misc.gperftools import Profiler
sage: Profiler()
Profiler logging to ...
```

`filename`()

Return the file name

OUTPUT:

String.

EXAMPLES:

```
sage: from sage.misc.gperftools import Profiler
sage: prof = Profiler()
sage: prof.filename()
'.../tmp_....perf'
```

`save`(*filename, cumulative=True, verbose=True*)

Save report to disk.

INPUT:

- `filename` – string. The filename to save at. Must end with one of `.dot`, `.ps`, `.pdf`, `.svg`, `.gif`, or `.txt` to specify the output file format.
- `cumulative` – boolean (optional, default: `True`). Whether to return cumulative timings.
- `verbose` – boolean (optional, default: `True`). Whether to print informational messages.

EXAMPLES:

```
sage: from sage.misc.gperftools import Profiler, run_100ms
sage: prof = Profiler()
sage: prof.start()     # optional - gperftools
sage: run_100ms()      # optional - gperftools
sage: prof.stop()      # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
sage: f = tmp_filename(ext='.txt')      # optional - gperftools
sage: prof.save(f, verbose=False)       # optional - gperftools
```

start()

Start profiling

EXAMPLES:

```
sage: from sage.misc.gperftools import Profiler, run_100ms
sage: prof = Profiler()
sage: prof.start()     # optional - gperftools
sage: run_100ms()
sage: prof.stop()      # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
```

stop()

Stop the CPU profiler

EXAMPLES:

```
sage: from sage.misc.gperftools import Profiler, run_100ms
sage: prof = Profiler()
sage: prof.start()     # optional - gperftools
sage: run_100ms()
sage: prof.stop()      # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
```

top(*cumulative=True*)

Print text report

OUTPUT:

Nothing. A textual report is printed to stdout.

EXAMPLES:

```
sage: from sage.misc.gperftools import Profiler
sage: prof = Profiler()
sage: prof.start()     # optional - gperftools
sage: # do something
sage: prof.stop()      # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
sage: prof.top()    # optional - gperftools
Using local file ...
Using local file ...
```

sage.misc.gperftools.**crun**(*s*, *evaluator*)

Profile single statement.

- `s` – string. Sage code to profile.
- `evaluator` – callable to evaluate.

EXAMPLES:

```
sage: import sage.misc.gperftools as gperf
sage: ev = lambda ex:eval(ex, globals(), locals())
sage: gperf.crun('gperf.run_100ms()', evaluator=ev)   # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
Using local file ...
Using local file ...
```

`sage.misc.gperftools.run_100ms()`

Used for doctesting.

A function that performs some computation for more than (but not that much more than) 100ms.

EXAMPLES:

```
sage: from sage.misc.gperftools import run_100ms
sage: run_100ms()
```

CHAPTER

FIFTY

LATEX

50.1 Installing and using SageTeX

SageTeX is a system for embedding computations and plots from Sage into LaTeX documents. It is included by default with Sage, so if you have installed Sage, you already have SageTeX. However, to get it to work, you need to make TeX aware of SageTeX. Instructions for that are in the "Make SageTeX known to TeX" section of the Sage installation guide (this link should take you to a local copy of the installation guide).

50.2 LaTeX printing support

In order to support latex formatting, an object should define a special method `_latex_(self)` that returns a string, which will be typeset in a mathematical mode (the exact mode depends on circumstances).

AUTHORS:

- William Stein: original implementation
- Joel B. Mohler: latex_variable_name() drastic rewrite and many doc-tests

class `sage.misc.latex.`**`Latex`**(*debug=False*, *slide=False*, *density=150*, *pdflatex=None*, *engine=None*)

Bases: `sage.misc.latex.LatexCall`

Enter, e.g.,

```
%latex
The equation $y^2 = x^3 + x$ defines an elliptic curve.
We have $2006 = \sage{factor(2006)}$.
```

in an input cell in the notebook to get a typeset version. Use `%latex_debug` to get debugging output.

Use `latex(...)` to typeset a Sage object. Use `LatexExpr` to typeset LaTeX code that you create by hand.

Use `%slide` instead to typeset slides.

> **Warning:** You must have dvipng (or dvips and convert) installed on your operating system, or this command won't work.

EXAMPLES:

```
sage: latex(x^20 + 1)
x^{20} + 1
sage: latex(FiniteField(25,'a'))
\Bold{F}_{5^{2}}
sage: latex("hello")
\text{\texttt{hello}}
```

```
sage: LatexExpr(r"\frac{x^2 - 1}{x + 1} = x - 1")
\frac{x^2 - 1}{x + 1} = x - 1
```

LaTeX expressions can be added; note that a space is automatically inserted:

```
sage: LatexExpr(r"y \neq") + latex(x^20 + 1)
y \neq x^{20} + 1
```

add_macro (*macro*)

Append to the string of extra LaTeX macros, for use with %latex, %html, and %mathjax.

INPUT:

- `macro` – string

EXAMPLES:

```
sage: latex.extra_macros()
''
sage: latex.add_macro("\\newcommand{\\foo}{bar}")
sage: latex.extra_macros()
'\\newcommand{\\foo}{bar}'
sage: latex.extra_macros("")  # restore to default
```

add_package_to_preamble_if_available (*package_name*)

Adds a `\usepackage{package_name}` instruction to the latex preamble if not yet present there, and if `package_name.sty` is available in the LaTeX installation.

INPUT:

- `package_name` – a string

See also:

- `add_to_preamble()`
- `has_file()`.

TESTS:

```
sage: latex.add_package_to_preamble_if_available("xypic")
sage: latex.add_package_to_preamble_if_available("nonexistent_package")
sage: latex.extra_preamble()       # optional - latex
'\\usepackage{xypic}\n'
sage: latex.extra_preamble('')
```

add_to_mathjax_avoid_list (*s*)

Add to the list of strings which signal that MathJax should not be used when 'view'ing.

INPUT:

- `s` – string; add `s` to the list of 'MathJax avoid' strings

If you want to replace the current list instead of adding to it, use `latex.mathjax_avoid_list`.

EXAMPLES:

```
sage: latex.add_to_mathjax_avoid_list("\\mathsf")
sage: latex.mathjax_avoid_list()  # display current setting
['\\mathsf']
sage: latex.add_to_mathjax_avoid_list("tkz-graph")
sage: latex.mathjax_avoid_list()  # display current setting
['\\mathsf', 'tkz-graph']
```

```
sage: latex.mathjax_avoid_list([])  # reset to default
sage: latex.mathjax_avoid_list()
[]
```

add_to_preamble(*s*)

Append to the string `s` of extra LaTeX macros, for use with %latex. Anything in this string won't be processed by %mathjax.

EXAMPLES:

```
sage: latex.extra_preamble()
''
sage: latex.add_to_preamble("\\DeclareMathOperator{\\Ext}{Ext}")
```

At this point, a notebook cell containing

```
%latex
$\Ext_A^*(\GF{2}, \GF{2}) \Rightarrow \pi_*^s*(S^0)$
```

will be typeset correctly.

```
sage: latex.add_to_preamble("\\usepackage{xypic}")
sage: latex.extra_preamble()
'\\DeclareMathOperator{\\Ext}{Ext}\\usepackage{xypic}'
```

Now one can put various xypic diagrams into a %latex cell, such as

```
%latex
\[ \xymatrix{ \circ \ar `r[d]^{a} `[rr]^{b} `/4pt[rr]^{c} `[rrr]^{d}
`_dl[drrr]^{e} [drrr]^{f} & \circ & \circ & \circ \\ \circ & \circ &
\circ & \circ } \]
```

Reset the preamble to its default, the empty string:

```
sage: latex.extra_preamble('')
sage: latex.extra_preamble()
''
```

blackboard_bold(*t=None*)

Controls whether Sage uses blackboard bold or ordinary bold face for typesetting ZZ, RR, etc.

INPUT:

- `t` – boolean or `None`

OUTPUT:

If `t` is `None`, return the current setting (`True` or `False`).

If `t` is `True`, use blackboard bold (`\mathbb`); otherwise use boldface (`\mathbf`).

EXAMPLES:

```
sage: latex.blackboard_bold()
False
sage: latex.blackboard_bold(True)
sage: latex.blackboard_bold()
True
sage: latex.blackboard_bold(False)
```

check_file(*file_name*, *more_info=''*)

INPUT:

- `file_name` – a string
- `more_info` – a string (default: "")

Emit a warning if the local LaTeX installation does not include `file_name`. The string `more_info` is appended to the warning message. The warning is only emitted the first time this method is called.

EXAMPLES:

```
sage: latex.check_file("article.cls")       # optional - latex
sage: latex.check_file("some_inexistent_file.sty")
Warning: `some_inexistent_file.sty` is not part of this computer's TeX installation.
sage: latex.check_file("some_inexistent_file.sty")
sage: latex.check_file("some_inexistent_file.sty", "This file is required for blah. It can b
Warning: `some_inexistent_file.sty` is not part of this computer's TeX installation.
This file is required for blah. It can be downloaded from: http://blah.org/
```

This test checks that the bug in trac ticket #9091 is fixed:

```
sage: latex.check_file("article.cls", "The article class is really critical.")    # optional
```

engine(*e=None*)

Set Sage to use `e` as latex engine when typesetting with `view()`, in `%latex` cells, etc.

INPUT:

- `e` – 'latex', 'pdflatex', 'xelatex' or `None`

If `e` is `None`, return the current engine.

If using the XeLaTeX engine, it will almost always be necessary to set the proper preamble with `extra_preamble()` or `add_to_preamble()`. For example:

```
latex.extra_preamble(r'''\usepackage{fontspec,xunicode,xltxtra}
\setmainfont[Mapping=tex-text]{some font here}
\setmonofont[Mapping=tex-text]{another font here}''')
```

EXAMPLES:

```
sage: latex.engine()
'pdflatex'
sage: latex.engine("latex")
sage: latex.engine()
'latex'
sage: latex.engine("xelatex")
sage: latex.engine()
'xelatex'
```

eval(*x, globals, strip=False, filename=None, debug=None, density=None, pdflatex=None, engine=None, locals={}*)

Compiles the formatted tex given by `x` as a png and writes the output file to the directory given by `filename`.

INPUT:

- `globals` – a globals dictionary
- `x` – string to evaluate.
- `strip` – ignored
- `filename` – output filename
- `debug` – whether to print verbose debugging output

- `density` – how big output image is.
- `pdflatex` – whether to use pdflatex. This is deprecated. Use `engine` option instead.
- `engine` – latex engine to use. Currently latex, pdflatex, and xelatex are supported.
- `locals` - extra local variables used when evaluating Sage code in `x`.

> **Warning:** When using latex (the default), you must have 'dvipng' (or 'dvips' and 'convert') installed on your operating system, or this command won't work. When using pdflatex or xelatex, you must have 'convert' installed.

OUTPUT:

If it compiled successfully, this returns an empty string `''`, otherwise it returns `None`.

EXAMPLES:

```
# This would generate a file named "test.png"
sage: latex.eval("\\ZZ[x]", locals(), filename="test") # not tested
''
# This would generate a file named "/path/to/test.png"
sage: latex.eval("\\ZZ[x]", locals(), filename="/path/to/test") # not tested
''
sage: latex.eval("\ThisIsAnInvalidCommand", {}) # optional -- ImageMagick
An error
...
No pages of output.
<BLANKLINE>
```

extra_macros (*macros=None*)

String containing extra LaTeX macros to use with %latex, %html, and %mathjax.

INPUT:

- `macros` – string (default: `None`)

If `macros` is `None`, return the current string. Otherwise, set it to `macros`. If you want to *append* to the string of macros instead of replacing it, use `latex.add_macro`.

EXAMPLES:

```
sage: latex.extra_macros("\\newcommand{\\foo}{bar}")
sage: latex.extra_macros()
'\\newcommand{\\foo}{bar}'
sage: latex.extra_macros("")
sage: latex.extra_macros()
''
```

extra_preamble (*s=None*)

String containing extra preamble to be used with %latex. Anything in this string won't be processed by %mathjax.

INPUT:

- `s` – string or `None`

If `s` is `None`, return the current preamble. Otherwise, set it to `s`. If you want to *append* to the current extra preamble instead of replacing it, use `latex.add_to_preamble`.

You will almost certainly need to use this when using the XeLaTeX engine; see below or the documentation for `engine()` for a suggested preamble.

EXAMPLES:

```
sage: latex.extra_preamble("\\DeclareMathOperator{\\Ext}{Ext}")
sage: latex.extra_preamble()
'\\DeclareMathOperator{\\Ext}{Ext}'
sage: latex.extra_preamble("\\"+r"usepackage{fontspec,xunicode,xltxtra}\setmainfont[Mapping=
sage: latex.extra_preamble()
'\\usepackage{fontspec,xunicode,xltxtra}\\setmainfont[Mapping=tex-text]{UnBatang}\\setmonofo
sage: latex.extra_preamble("")
sage: latex.extra_preamble()
''
```

has_file (*file_name*)

INPUT:

- `file_name` – a string

Tests whether the local LaTeX installation includes `file_name`.

EXAMPLES:

```
sage: latex.has_file("article.cls")      # optional - latex
True
sage: latex.has_file("some_inexistent_file.sty")
False
```

mathjax_avoid_list (*L=None*)

List of strings which signal that MathJax should not be used when 'view'ing.

INPUT:

- `L` – A list or `None`

If `L` is `None`, then return the current list. Otherwise, set it to `L`. If you want to *append* to the current list instead of replacing it, use `latex.add_to_mathjax_avoid_list`.

EXAMPLES:

```
sage: latex.mathjax_avoid_list(["\\mathsf", "pspicture"])
sage: latex.mathjax_avoid_list()  # display current setting
['\\mathsf', 'pspicture']
sage: latex.mathjax_avoid_list([])  # reset to default
sage: latex.mathjax_avoid_list()
[]
```

matrix_column_alignment (*align=None*)

Changes the column-alignment of the LaTeX representation of matrices.

INPUT:

- `align` - a string (`'r'` for right, `'c'` for center, `'l'` for left) or `None`.

OUTPUT:

If `align` is `None`, then returns the current alignment-string. Otherwise, set this alignment.

The input `align` can be any string which the LaTeX `array`-environment understands as a parameter for aligning a column.

EXAMPLES:

```
sage: a = matrix(1, 1, [42])
sage: latex(a)
\left(\begin{array}{r}
42
\end{array}\right)
```

```
sage: latex.matrix_column_alignment('c')
sage: latex(a)
\left(\begin{array}{c}
42
\end{array}\right)
sage: latex.matrix_column_alignment('l')
sage: latex(a)
\left(\begin{array}{l}
42
\end{array}\right)
```

Restore defaults:

```
sage: latex.matrix_column_alignment('r')
```

matrix_delimiters (*left=None*, *right=None*)

Change the left and right delimiters for the LaTeX representation of matrices

INPUT:

- `left`, `right` - strings or `None`

If both `left` and `right` are `None`, then return the current delimiters. Otherwise, set the left and/or right delimiters, whichever are specified.

Good choices for `left` and `right` are any delimiters which LaTeX understands and knows how to resize; some examples are:

- parentheses: '(', ')'
- brackets: '[', ']'
- braces: '\{', '\}'
- vertical lines: '|'
- angle brackets: '\langle', '\rangle'

Note: Putting aside aesthetics, you may combine these in any way imaginable; for example, you could set `left` to be a right-hand bracket ']' and `right` to be a right-hand brace '\}', and it will be typeset correctly.

EXAMPLES:

```
sage: a = matrix(1, 1, [17])
sage: latex(a)
\left(\begin{array}{r}
17
\end{array}\right)
sage: latex.matrix_delimiters("[", "]")
sage: latex(a)
\left[\begin{array}{r}
17
\end{array}\right]
sage: latex.matrix_delimiters(left="\\{")
sage: latex(a)
\left\{\begin{array}{r}
17
\end{array}\right]
sage: latex.matrix_delimiters()
['\\{', ']']
```

Restore defaults:

```
sage: latex.matrix_delimiters("(", ")")
```

vector_delimiters(*left=None*, *right=None*)

Change the left and right delimiters for the LaTeX representation of vectors

INPUT:

- `left`, `right` – strings or `None`

If both `left` and `right` are `None`, then return the current delimiters. Otherwise, set the left and/or right delimiters, whichever are specified.

Good choices for `left` and `right` are any delimiters which LaTeX understands and knows how to resize; some examples are:

- parentheses: '(', ')'
- brackets: '[', ']'
- braces: '\{', '\}'
- vertical lines: '|'
- angle brackets: '\langle', '\rangle'

Note: Putting aside aesthetics, you may combine these in any way imaginable; for example, you could set `left` to be a right-hand bracket ']' and `right` to be a right-hand brace '\}', and it will be typeset correctly.

EXAMPLES:

```
sage: a = vector(QQ, [1,2,3])
sage: latex(a)
\left(1,\,2,\,3\right)
sage: latex.vector_delimiters("[", "]")
sage: latex(a)
\left[1,\,2,\,3\right]
sage: latex.vector_delimiters(right="\\}")
sage: latex(a)
\left[1,\,2,\,3\right\}
sage: latex.vector_delimiters()
['[', '\\}']
```

Restore defaults:

```
sage: latex.vector_delimiters("(", ")")
```

class sage.misc.latex.**LatexCall**

Typeset Sage objects via a `__call__` method to this class, typically by calling those objects' `_latex_` methods. The class `Latex` inherits from this. This class is used in `latex_macros`, while functions from `latex_macros` are used in `Latex`, so this is here primarily to avoid circular imports.

EXAMPLES:

```
sage: from sage.misc.latex import LatexCall
sage: LatexCall()(ZZ)
\Bold{Z}
sage: LatexCall().__call__(ZZ)
\Bold{Z}
```

This returns an instance of the class `LatexExpr`:

```
sage: type(LatexCall()(ZZ))
<class 'sage.misc.latex.LatexExpr'>
```

class `sage.misc.latex.`**`LatexExamples`**

A catalogue of Sage objects with complicated `_latex_` methods. Use these for testing `latex()`, `view()`, the Typeset button in the notebook, etc.

The classes here only have `__init__`, `_repr_`, and `_latex_` methods.

EXAMPLES:

```
sage: from sage.misc.latex import latex_examples
sage: K = latex_examples.knot()
sage: K
LaTeX example for testing display of a knot produced by xypic...
sage: latex(K)
\vtop{\vbox{\xygraph{!{0;/r1.5pc/:}
[u] !{\vloop<(-.005)\khole||\vcrossneg \vunder- }
[] !{\ar @{-}@'{p-(1,0)@+}+(-1,1)}
[ul] !{\vcap[3]>\khole}
[rrr] !{\ar @{-}@'{p-(0,1)@+}-(1,1)}
}}}
```

class **`diagram`**

Bases: `sage.structure.sage_object.SageObject`

LaTeX example for testing display of commutative diagrams. See its string representation for details.

EXAMPLES:

```
sage: from sage.misc.latex import latex_examples
sage: CD = latex_examples.diagram()
sage: CD
LaTeX example for testing display of a commutative diagram...
```

class `LatexExamples.`**`graph`**

Bases: `sage.structure.sage_object.SageObject`

LaTeX example for testing display of graphs. See its string representation for details.

EXAMPLES:

```
sage: from sage.misc.latex import latex_examples
sage: G = latex_examples.graph()
sage: G
LaTeX example for testing display of graphs...
```

class `LatexExamples.`**`knot`**

Bases: `sage.structure.sage_object.SageObject`

LaTeX example for testing display of knots. See its string representation for details.

EXAMPLES:

```
sage: from sage.misc.latex import latex_examples
sage: K = latex_examples.knot()
sage: K
LaTeX example for testing display of a knot...
```

class `LatexExamples.`**`pstricks`**

Bases: `sage.structure.sage_object.SageObject`

LaTeX example for testing display of pstricks output. See its string representation for details.

EXAMPLES:

```
sage: from sage.misc.latex import latex_examples
sage: PS = latex_examples.pstricks()
sage: PS
LaTeX example for testing display of pstricks...
```

class sage.misc.latex.**LatexExpr**

Bases: str

A class for LaTeX expressions.

Normally, objects of this class are created by a latex() call. It is also possible to generate LatexExpr directly from a string, which must contain valid LaTeX code for typesetting in math mode (without dollar signs). In the Sage notebook, use pretty_print() or the "Typeset" checkbox to actually see the typeset LaTeX code; alternatively, from either the command-line or the notebook, use the view() function.

INPUT:

- str – a string with valid math mode LaTeX code (or something which can be converted to such a string).

OUTPUT:

- LatexExpr wrapping the string representation of the input.

EXAMPLES:

```
sage: latex(x^20 + 1)
x^{20} + 1
sage: LatexExpr(r"\frac{x^2 + 1}{x - 2}")
\frac{x^2 + 1}{x - 2}
```

LatexExpr simply converts to string without doing anything extra, it does *not* call latex():

```
sage: latex(ZZ)
\Bold{Z}
sage: LatexExpr(ZZ)
Integer Ring
```

The result of latex() is of type LatexExpr:

```
sage: L = latex(x^20 + 1)
sage: L
x^{20} + 1
sage: type(L)
<class 'sage.misc.latex.LatexExpr'>
```

A LatexExpr can be converted to a plain string:

```
sage: str(latex(x^20 + 1))
'x^{20} + 1'
```

class sage.misc.latex.**MathJax**

Render LaTeX input using MathJax. This returns a MathJaxExpr.

EXAMPLES:

```
sage: from sage.misc.latex import MathJax
sage: MathJax()(3)
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}3</script></html>
sage: MathJax()(ZZ)
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}\Bold{Z}</script><
```

eval (*x*, *globals=None*, *locals=None*, *mode='display'*, *combine_all=False*)

Render LaTeX input using MathJax. This returns a `MathJaxExpr`.

INPUT:

- `x` - a Sage object
- `globals` - a globals dictionary
- `locals` - extra local variables used when evaluating Sage code in `x`.
- **`mode` - string (optional, default `'display'`):** `'display'` for displaymath, `'inline'` for inline math, or `'plain'` for just the LaTeX code without the surrounding html and script tags.
- `combine_all` - boolean (Default: `False`): If `combine_all` is `True` and the input is a tuple, then it does not return a tuple and instead returns a string with all the elements separated by a single space.

OUTPUT:

A `MathJaxExpr`

EXAMPLES:

```
sage: from sage.misc.latex import MathJax
sage: MathJax().eval(3, mode='display')
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}3</script></ht
sage: MathJax().eval(3, mode='inline')
<html><script type="math/tex">\newcommand{\Bold}[1]{\mathbf{#1}}3</script></html>
sage: MathJax().eval(type(3), mode='inline')
<html><script type="math/tex">\newcommand{\Bold}[1]{\mathbf{#1}}\verb|<type|\phantom{\verb!x
```

class sage.misc.latex.**MathJaxExpr** (*y*)

An arbitrary MathJax expression that can be nicely concatenated.

EXAMPLES:

```
sage: from sage.misc.latex import MathJaxExpr
sage: MathJaxExpr("a^{2}") + MathJaxExpr("x^{-1}")
a^{2}x^{-1}
```

sage.misc.latex.**None_function** (*x*)

Returns the LaTeX code for `None`.

INPUT:

- `x` – `None`

EXAMPLES:

```
sage: from sage.misc.latex import None_function
sage: print None_function(None)
\mathrm{None}
```

sage.misc.latex.**bool_function** (*x*)

Returns the LaTeX code for a boolean `x`.

INPUT:

- `x` – boolean

EXAMPLES:

```
sage: from sage.misc.latex import bool_function
sage: print bool_function(2==3)
\mathrm{False}
sage: print bool_function(3==(2+1))
\mathrm{True}
```

`sage.misc.latex.`**`builtin_constant_function`**`(x)`

Returns the LaTeX code for a builtin constant `x`.

INPUT:

- `x` – builtin constant

See also:

Python built-in Constants http://docs.python.org/library/constants.html

EXAMPLES:

```
sage: from sage.misc.latex import builtin_constant_function
sage: builtin_constant_function(True)
'\\mbox{\\rm True}'
sage: builtin_constant_function(None)
'\\mbox{\\rm None}'
sage: builtin_constant_function(NotImplemented)
'\\mbox{\\rm NotImplemented}'
sage: builtin_constant_function(Ellipsis)
'\\mbox{\\rm Ellipsis}'
```

TESTS:

```
sage: sage.misc.latex.EMBEDDED_MODE = True
sage: builtin_constant_function(True)
'{\\rm True}'
sage: sage.misc.latex.EMBEDDED_MODE = False
```

`sage.misc.latex.`**`coeff_repr`**`(c)`

LaTeX string representing coefficients in a linear combination.

INPUT:

- `c` – a coefficient (i.e., an element of a ring)

OUTPUT:

A string

EXAMPLES:

```
sage: from sage.misc.latex import coeff_repr
sage: coeff_repr(QQ(1/2))
'\\frac{1}{2}'
sage: coeff_repr(-x^2)
'\\left(-x^{2}\\right)'
```

`sage.misc.latex.`**`dict_function`**`(x)`

Returns the LaTeX code for a dictionary `x`.

INPUT:

- `x` – a dictionary

EXAMPLES:

```
sage: from sage.misc.latex import dict_function
sage: x,y,z = var('x,y,z')
sage: print dict_function({x/2: y^2})
\left\{\frac{1}{2} \, x : y^{2}\right\}
sage: d = {(1,2,x^2): [sin(z^2), y/2]}
sage: latex(d)
\left\{\left(1, 2, x^{2}\right) :
       \left[\sin\left(z^{2}\right), \frac{1}{2} \, y\right]\right\}
```

`sage.misc.latex.`**`float_function`**`(x)`

Returns the LaTeX code for a python float `x`.

INPUT:

- `x` – a python float

EXAMPLES:

```
sage: from sage.misc.latex import float_function
sage: float_function(float(3.14))
3.14
sage: float_function(float(1e-10))
1 \times 10^{-10}
sage: float_function(float(2e10))
20000000000.0
```

TESTS:

Check that trac ticket #7356 is fixed:

```
sage: latex(float(2e-13))
2 \times 10^{-13}
```

`sage.misc.latex.`**`has_latex_attr`**`(x)`

Return `True` if `x` has a `_latex_` attribute, except if `x` is a `type`, in which case return `False`.

EXAMPLES:

```
sage: from sage.misc.latex import has_latex_attr
sage: has_latex_attr(identity_matrix(3))
True
sage: has_latex_attr("abc")   # strings have no _latex_ method
False
```

Types inherit the `_latex_` method of the class to which they refer, but calling it is broken:

```
sage: T = type(identity_matrix(3)); T
<type 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
sage: hasattr(T, '_latex_')
True
sage: T._latex_()
Traceback (most recent call last):
...
TypeError: descriptor '_latex_' of 'sage.matrix.matrix0.Matrix' object needs an argument
sage: has_latex_attr(T)
False
```

`sage.misc.latex.`**`have_convert`**`()`

Return `True` if this computer has the program `convert`.

If this computer doesn't have convert installed, you may obtain it (along with the rest of the ImageMagick suite) from http://www.imagemagick.org

EXAMPLES:

```
sage: from sage.misc.latex import have_convert
sage: have_convert() # random
True
```

`sage.misc.latex.`**`have_dvipng`**`()`

Return `True` if this computer has the program `dvipng`.

If this computer doesn't have dvipng installed, you may obtain it from http://sourceforge.net/projects/dvipng/

EXAMPLES:

```
sage: from sage.misc.latex import have_dvipng
sage: have_dvipng() # random
True
```

`sage.misc.latex.`**`have_latex`**`()`

Return `True` if this computer has the program `latex`.

If this computer doesn't have LaTeX installed, you may obtain it from http://ctan.org/.

EXAMPLES:

```
sage: from sage.misc.latex import have_latex
sage: have_latex() # random
True
```

`sage.misc.latex.`**`have_pdflatex`**`()`

Return `True` if this computer has the program `pdflatex`.

If this computer doesn't have pdflatex installed, you may obtain it from http://ctan.org/.

EXAMPLES:

```
sage: from sage.misc.latex import have_pdflatex
sage: have_pdflatex() # random
True
```

`sage.misc.latex.`**`have_xelatex`**`()`

Return `True` if this computer has the program `xelatex`.

If this computer doesn't have xelatex installed, you may obtain it from http://ctan.org/.

EXAMPLES:

```
sage: from sage.misc.latex import have_xelatex
sage: have_xelatex() # random
True
```

`sage.misc.latex.`**`latex`**`(`*x*, *combine_all=False*`)`

Return a `LatexExpr` built out of the argument `x`.

INPUT:

- `x` – a Sage object
- `combine_all` – boolean (Default: `False`) If `combine_all` is `True` and the input is a tuple, then it does not return a tuple and instead returns a string with all the elements separated by a single space.

OUTPUT:

A `LatexExpr` built from `x`

EXAMPLES:

```
sage: latex(Integer(3))  # indirect doctest
3
sage: latex(1==0)
\mathrm{False}
sage: print latex([x,2])
\left[x, 2\right]
```

Check that trac ticket #11775 is fixed:

```
sage: latex((x,2), combine_all=True)
x 2
```

sage.misc.latex.**latex_extra_preamble**()

Return the string containing the user-configured preamble, `sage_latex_macros`, and any user-configured macros. This is used in the `eval()` method for the `Latex` class, and in `_latex_file_()`; it follows either `LATEX_HEADER` or `SLIDE_HEADER` (defined at the top of this file) which is a string containing the documentclass and standard usepackage commands.

EXAMPLES:

```
sage: from sage.misc.latex import latex_extra_preamble
sage: print latex_extra_preamble()
...

\newcommand{\ZZ}{\Bold{Z}}
\newcommand{\NN}{\Bold{N}}
\newcommand{\RR}{\Bold{R}}
\newcommand{\CC}{\Bold{C}}
\newcommand{\QQ}{\Bold{Q}}
\newcommand{\QQbar}{\overline{\QQ}}
\newcommand{\GF}[1]{\Bold{F}_{#1}}
\newcommand{\Zp}[1]{\ZZ_{#1}}
\newcommand{\Qp}[1]{\QQ_{#1}}
\newcommand{\Zmod}[1]{\ZZ/#1\ZZ}
\newcommand{\CDF}{\Bold{C}}
\newcommand{\CIF}{\Bold{C}}
\newcommand{\CLF}{\Bold{C}}
\newcommand{\RDF}{\Bold{R}}
\newcommand{\RIF}{\Bold{I} \Bold{R}}
\newcommand{\RLF}{\Bold{R}}
\newcommand{\CFF}{\Bold{CFF}}
\newcommand{\Bold}[1]{\mathbf{#1}}
```

sage.misc.latex.**latex_variable_name**(*x*, *is_fname=False*)

Return latex version of a variable name.

Here are some guiding principles for usage of this function:

1. If the variable is a single letter, that is the latex version.
2. If the variable name is suffixed by a number, we put the number in the subscript.
3. If the variable name contains an '_' we start the subscript at the underscore. Note that #3 trumps rule #2.
4. If a component of the variable is a Greek letter, escape it properly.
5. Recurse nicely with subscripts.

Refer to the examples section for how these rules might play out in practice.

EXAMPLES:

```
sage: from sage.misc.latex import latex_variable_name
sage: latex_variable_name('a')
'a'
sage: latex_variable_name('abc')
'\\mathit{abc}'
sage: latex_variable_name('sigma')
'\\sigma'
sage: latex_variable_name('sigma_k')
'\\sigma_{k}'
sage: latex_variable_name('sigma389')
'\\sigma_{389}'
sage: latex_variable_name('beta_00')
'\\beta_{00}'
sage: latex_variable_name('Omega84')
'\\Omega_{84}'
sage: latex_variable_name('sigma_alpha')
'\\sigma_{\\alpha}'
sage: latex_variable_name('nothing1')
'\\mathit{nothing}_{1}'
sage: latex_variable_name('nothing1', is_fname=True)
'{\\rm nothing}_{1}'
sage: latex_variable_name('nothing_abc')
'\\mathit{nothing}_{\\mathit{abc}}'
sage: latex_variable_name('nothing_abc', is_fname=True)
'{\\rm nothing}_{{\\rm abc}}'
sage: latex_variable_name('alpha_beta_gamma12')
'\\alpha_{\\beta_{\\gamma_{12}}}'
sage: latex_variable_name('x_ast')
'x_{\\ast}'
```

TESTS:

```
sage: latex_variable_name('_C')  # :trac:`16007`
'C'
sage: latex_variable_name('_K1')
'K_{1}'
```

`sage.misc.latex.`**`latex_varify`**(*a*, *is_fname=False*)

Convert a string `a` to a LaTeX string: if it's an element of `common_varnames`, then prepend a backslash. If `a` consists of a single letter, then return it. Otherwise, return either "{\rm a}" or "\mbox{a}" if "is_fname" flag is `True` or `False`.

INPUT:

- a – string

OUTPUT:

A string

EXAMPLES:

```
sage: from sage.misc.latex import latex_varify
sage: latex_varify('w')
'w'
sage: latex_varify('aleph')
'\\mathit{aleph}'
sage: latex_varify('aleph', is_fname=True)
'{\\rm aleph}'
sage: latex_varify('alpha')
```

```
'\\alpha'
sage: latex_varify('ast')
'\\ast'
```

TESTS:

sage: abc = var('abc') sage: latex((abc/(abc+1)+42)/(abc-1)) # trac #15870 frac{frac{mathit{abc}}{mathit{abc} + 1} + 42}{mathit{abc} - 1}

`sage.misc.latex.list_function(x)`

Returns the LaTeX code for a list `x`.

INPUT: `x` - a list

EXAMPLES:

```
sage: from sage.misc.latex import list_function
sage: list_function([1,2,3])
'\\left[1, 2, 3\\right]'
sage: latex([1,2,3])  # indirect doctest
\left[1, 2, 3\right]
sage: latex([Matrix(ZZ,3,range(9)), Matrix(ZZ,3,range(9))]) # indirect doctest
\left[\left(\begin{array}{rrr}
0 & 1 & 2 \\
3 & 4 & 5 \\
6 & 7 & 8
\end{array}\right), \left(\begin{array}{rrr}
0 & 1 & 2 \\
3 & 4 & 5 \\
6 & 7 & 8
\end{array}\right)\right]
```

`sage.misc.latex.png(x, filename, density=150, debug=False, do_in_background=False, tiny=False, pdflatex=True, engine='pdflatex')`

Create a png image representation of `x` and save to the given filename.

INPUT:

- `x` – object to be displayed
- `filename` – file in which to save the image
- `density` – integer (default: 150)
- `debug` – bool (default: `False`): print verbose output
- `do_in_background` – bool (default: `False`): Unused, kept for backwards compatibility
- `tiny` – bool (default: `False`): use 'tiny' font
- `pdflatex` – bool (default: `True`): use pdflatex. This option is deprecated. Use `engine` option instead. See below.
- `engine` – (default: `'pdflatex'`) `'latex'`, `'pdflatex'`, or `'xelatex'`

EXAMPLES:

```
sage: from sage.misc.latex import png
sage: png(ZZ[x], os.path.join(SAGE_TMP, "zz.png")) # random - error if no latex
```

`sage.misc.latex.pretty_print(*args)`

Try to pretty print the arguments in an intelligent way. For graphics objects, this returns their default representation. For other objects, in the notebook, this calls the `view()` command, while from the command line, this produces an html string suitable for processing by MathJax.

INPUT:

- `objects` – The input can be any Sage object, a list or tuple of Sage objects, or Sage objects passed in as separate arguments.

This function is used in the notebook when the "Typeset" button is checked.

EXAMPLES:

```
sage: pretty_print(ZZ)  # indirect doctest
<html><script type="math/tex">\newcommand{\Bold}[1]{\mathbf{#1}}\Bold{Z}</script></html>
sage: pretty_print("Integers = ", ZZ) # trac 11775
<html><script type="math/tex">\newcommand{\Bold}[1]{\mathbf{#1}}\verb|Integers|\phantom{\verb!x!
```

To typeset LaTeX code as-is, use `LatexExpr`:

```
sage: pretty_print(LatexExpr(r"\frac{x^2 + 1}{x - 2}"))
<html><script type="math/tex">\newcommand{\Bold}[1]{\mathbf{#1}}\frac{x^2 + 1}{x - 2}</script></
```

sage.misc.latex.**pretty_print_default**(*enable=True*)

Enable or disable default pretty printing. Pretty printing means rendering things so that MathJax or some other latex-aware front end can render real math.

This function is pretty useless without the notebook, it shoudn't be in the global namespace.

INPUT:

- `enable` – bool (optional, default `True`). If `True`, turn on pretty printing; if `False`, turn it off.

EXAMPLES:

```
sage: pretty_print_default(True)
sage: 'foo'
\newcommand{\Bold}[1]{\mathbf{#1}}\verb|foo|
sage: pretty_print_default(False)
sage: 'foo'
'foo'
```

sage.misc.latex.**print_or_typeset**(*object*)

'view' or 'print' the object depending on the situation.

In particular, if in notebook mode with the typeset box checked, view the object. Otherwise, print the object.

INPUT:

- `object` – Anything

EXAMPLES:

```
sage: sage.misc.latex.print_or_typeset(3)
3
sage: sage.misc.latex.EMBEDDED_MODE=True
sage: sage.misc.latex.print_or_typeset(3)
3
sage: TEMP = sys.displayhook
sage: sys.displayhook = sage.misc.latex.pretty_print
sage: sage.misc.latex.print_or_typeset(3)
<html><script type="math/tex">\newcommand{\Bold}[1]{\mathbf{#1}}3</script></html>
sage: sage.misc.latex.EMBEDDED_MODE=False
sage: sys.displayhook = TEMP
```

sage.misc.latex.**repr_lincomb**(*symbols*, *coeffs*)

Compute a latex representation of a linear combination of some formal symbols.

INPUT:

- `symbols` – list of symbols
- `coeffs` – list of coefficients of the symbols

OUTPUT:

A string

EXAMPLES:

```
sage: t = PolynomialRing(QQ, 't').0
sage: from sage.misc.latex import repr_lincomb
sage: repr_lincomb(['a', 's', ''], [-t, t - 2, t^12 + 2])
'-t\\text{\\texttt{a}} + \\left(t - 2\\right)\\text{\\texttt{s}} + \\left(t^{12} + 2\\right)'
sage: repr_lincomb(['a', 'b'], [1,1])
'\\text{\\texttt{a}} + \\text{\\texttt{b}}'
```

Verify that a certain corner case works (see trac ticket #5707 and trac ticket #5766):

```
sage: repr_lincomb([1,5,-3],[2,8/9,7])
'2\\cdot 1 + \\frac{8}{9}\\cdot 5 + 7\\cdot -3'
```

sage.misc.latex.**str_function**(*x*)

Return a LaTeX representation of the string `x`.

The main purpose of this function is to generate LaTeX representation for classes that do not provide a customized method.

If `x` contains only digits with, possibly, a single decimal point and/or a sign in front, it is considered to be its own representation. Otherwise each line of `x` is wrapped in a `\texttt` command and these lines are assembled in a left-justified array. This gives to complicated strings the closest look to their "terminal representation".

> **Warning:** Such wrappers **cannot** be used as arguments of LaTeX commands or in command definitions. If this causes you any problems, they probably can be solved by implementing a suitable `_latex_` method for an appropriate class.

INPUT:

- `x` – a string.

OUTPUT:

A string

EXAMPLES:

```
sage: from sage.misc.latex import str_function
sage: str_function('34')
'34'
sage: str_function('34.5')
'34.5'
sage: str_function('-34.5')
'-34.5'
sage: str_function('+34.5')
'+34.5'
sage: str_function('hello_world')
'\\text{\\texttt{hello{\\char`\\_}world}}'
```

```
sage: str_function('-1.00000?') # trac 12178
'-1.00000?'
```

sage.misc.latex.**tuple_function**(*x*, *combine_all=False*)

Returns the LaTeX code for a tuple `x`.

INPUT:

- `x` – a tuple
- `combine_all` – boolean (Default: `False`) If `combine_all` is `True`, then it does not return a tuple and instead returns a string with all the elements separated by a single space. It does not collapse tuples which are inside tuples.

EXAMPLES:

```
sage: from sage.misc.latex import tuple_function
sage: tuple_function((1,2,3))
'\\left(1, 2, 3\\right)'
```

Check that trac ticket #11775 is fixed:

```
sage: tuple_function((1,2,3), combine_all=True)
'1 2 3'
sage: tuple_function(((1,2),3), combine_all=True)
'\\left(1, 2\\right) 3'
```

sage.misc.latex.**view**(*objects*, *title='Sage'*, *debug=False*, *sep=''*, *tiny=False*, *pdflatex=None*, *engine=None*, *viewer=None*, *tightpage=None*, *mode='inline'*, *combine_all=False*, ***kwds*)

Compute a latex representation of each object in objects, compile, and display typeset. If used from the command line, this requires that latex be installed.

INPUT:

- `objects` – list (or object)
- `title` – string (default: `'Sage'`): title for the document
- `debug` – bool (default: `False`): print verbose output
- `sep` – string (default: ‘’): separator between math objects
- `tiny` – bool (default: `False`): use tiny font.
- `pdflatex` – bool (default: `False`): use pdflatex. This is deprecated. Use `'engine'` option instead.
- `engine` – string or `None` (default: `None`). Can take the following values:
  - `None` – the value defined in the LaTeX global preferences `latex.engine()` is used.
  - `'pdflatex'` – compilation does tex -> pdf
  - `'xelatex'` – compilation does tex -> pdf
  - `'latex'` – compilation first tries tex -> dvi -> png and if an error occurs then tries dvi -> ps -> pdf. This is slower than `'pdflatex'` and known to be broken when overfull hbox are detected.
- `viewer` – string or `None` (default: `None`): specify a viewer to use; currently the only options are `None` and `'pdf'`.
- `tightpage` – bool (default: `False`): use the LaTeX package ‘preview’ with the ‘tightpage’ option.
- `mode` – string (default: `'inline'`): `'display'` for displaymath or `'inline'` for inline math

- `combine_all` – bool (default: `False`): If `combine_all` is `True` and the input is a tuple, then it does not return a tuple and instead returns a string with all the elements separated by a single space.

OUTPUT:

Display typeset objects.

This function behaves differently depending on whether in notebook mode or not.

If not in notebook mode, the output is displayed in a separate viewer displaying a dvi (or pdf) file, with the following: the title string is printed, centered, at the top. Beneath that, each object in `objects` is typeset on its own line, with the string `sep` inserted between these lines.

The value of `sep` is inserted between each element of the list `objects`; you can, for example, add vertical space between objects with `sep='\\vspace{15mm}'`, while `sep='\\hrule'` adds a horizontal line between objects, and `sep='\\newpage'` inserts a page break between objects.

If `pdflatex` is `True`, then the latex engine is set to pdflatex.

If the `engine` is either `pdflatex` or `xelatex`, it produces a pdf file. Otherwise, it produces a dvi file, and if the program dvipng is installed, it checks the dvi file by trying to convert it to a png file. If this conversion fails, the dvi file probably contains some postscript special commands or it has other issues which might make displaying it a problem; in this case, the file is converted to a pdf file, which is then displayed.

Setting `viewer` to `'pdf'` forces the use of a separate viewer, even in notebook mode. This also sets the latex engine to be `pdflatex` if the current engine is latex.

Setting the option `tightpage` to `True` tells LaTeX to use the package 'preview' with the 'tightpage' option. Then, each object is typeset in its own page, and that page is cropped to exactly the size of the object. This is typically useful for very large pictures (like graphs) generated with tikz. This only works when using a separate viewer. Note that the object are currently typeset in plain math mode rather than displaymath, because the latter imposes a limit on the width of the picture. Technically, `tightpage` adds

```
\\usepackage[tightpage,active]{preview}
\\PreviewEnvironment{page}
```

to the LaTeX preamble, and replaces the `\\[` and `\\]` around each object by `\\begin{page}$` and `$\\end{page}`.

If in notebook mode with `viewer` equal to `None`, this usually uses MathJax – see the next paragraph for the exception – to display the output in the notebook. Only the first argument, `objects`, is relevant; the others are ignored. If `objects` is a list, each object is printed on its own line.

In the notebook, this *does not* use MathJax if the LaTeX code for `objects` contains a string in `latex.mathjax_avoid_list()`. In this case, it creates and displays a png file.

EXAMPLES:

```
sage: sage.misc.latex.EMBEDDED_MODE = True
sage: view(3)
<html><script type="math/tex">\newcommand{\Bold}[1]{\mathbf{#1}}3</script></html>
sage: view(3, mode='display')
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}3</script></html>
sage: view((x,2), combine_all=True) # trac 11775
<html><script type="math/tex">\newcommand{\Bold}[1]{\mathbf{#1}}x 2</script></html>
sage: sage.misc.latex.EMBEDDED_MODE = False
```

TESTS:

```
sage: from sage.misc.latex import _run_latex_, _latex_file_
sage: g = sage.misc.latex.latex_examples.graph()
sage: latex.add_to_preamble(r"\usepackage{tkz-graph}")
sage: file = os.path.join(SAGE_TMP, "temp.tex")
```

```
sage: O = open(file, 'w'); O.write(_latex_file_(g)); O.close()
sage: _run_latex_(file, engine="pdflatex") # optional - latex
'pdf'

sage: latex.extra_preamble('') # reset the preamble

sage: view(4, engine="garbage")
Traceback (most recent call last):
...
ValueError: Unsupported LaTeX engine.
sage: sage.misc.latex.EMBEDDED_MODE = True
sage: view(4, engine="garbage", viewer="pdf")
Traceback (most recent call last):
...
ValueError: Unsupported LaTeX engine.
```

50.3 LaTeX macros

AUTHORS:

- John H. Palmieri (2009-03)

The code here sets up LaTeX macro definitions for use in the documentation. To add a macro, modify the list `macros`, near the end of this file, and then run 'sage -b'. The entries in this list are used to produce `sage_latex_macros`, a list of strings of the form '\newcommand...', and `sage_mathjax_macros`, a list of strings suitable for parsing by MathJax. The LaTeX macros are produced using the `_latex_` method for each Sage object listed in `macros`, and the MathJax macros are produced from the LaTeX macros. The list of LaTeX macros is used in the file `SAGE_DOC/common/conf.py` to add to the preambles of both the LaTeX file used to build the PDF version of the documentation and the LaTeX file used to build the HTML version. The list of MathJax macros is used in the file `sagenb/notebook/tutorial.py` to define MathJax macros for use in the live documentation (and also in the notebook).

Any macro defined here may be used in docstrings or in the tutorial (or other pieces of documentation). In a docstring, for example, "ZZ" in backquotes (demarking math mode) will appear as "ZZ" in interactive help, but will be typeset as "\Bold{Z}" in the reference manual.

More details on the list `macros`: the entries are lists or tuples of the form `[name]` or `[name, arguments]`, where `name` is a string and `arguments` consists of valid arguments for the Sage object named `name`. For example, `["ZZ"]` and `["GF", 2]` produce the LaTeX macros '\newcommand{\ZZ}{\Bold{Z}}' and '\newcommand{\GF}[1]{\Bold{F}_{#1}}', respectively. (For the second of these, `latex(GF(2))` is called and the string '2' gets replaced by '#1', so `["GF", 17]` would have worked just as well. `["GF", p]` would have raised an error, though, because `p` is not defined, and `["GF", 4]` would have raised an error, because to define the field with four elements in Sage, you also need to specify the name of a generator.)

To see evidence of the results of the code here, run `sage -docbuild tutorial latex` (for example), and look at the resulting LaTeX file in `SAGE_DOC/output/latex/en/tutorial/`. The preamble should contain 'newcommand' lines for each of the entries in `macros`.

`sage.misc.latex_macros.`**`convert_latex_macro_to_mathjax`**(*macro*)

This converts a LaTeX macro definition (newcommand...) to a MathJax macro definition (MathJax.Macro...).

INPUT:

- `macro` - LaTeX macro definition

See the web page http://www.mathjax.org/docs/1.1/options/TeX.html for a description of the format for MathJax macros.

EXAMPLES:

```
sage: from sage.misc.latex_macros import convert_latex_macro_to_mathjax
sage: convert_latex_macro_to_mathjax('\\newcommand{\\ZZ}{\\Bold{Z}}')
'ZZ: "\\\\Bold{Z}"'
sage: convert_latex_macro_to_mathjax('\\newcommand{\\GF}[1]{\\Bold{F}_{#1}}')
'GF: ["\\\\Bold{F}_{#1}",1]'
```

`sage.misc.latex_macros.`**`produce_latex_macro`**`(`*name*, **sample_args*`)`

Produce a string defining a LaTeX macro.

INPUT:

- `name` - name of macro to be defined, also name of corresponding Sage object
- `sample_args` - (optional) sample arguments for this Sage object

EXAMPLES:

```
sage: from sage.misc.latex_macros import produce_latex_macro
sage: produce_latex_macro('ZZ')
'\\newcommand{\\ZZ}{\\Bold{Z}}'
```

If the Sage object takes arguments, then the LaTeX macro will accept arguments as well. You must pass valid arguments, which will then be converted to #1, #2, etc. in the macro definition. The following allows the use of "GF{p^n}", for example:

```
sage: produce_latex_macro('GF', 37)
'\\newcommand{\\GF}[1]{\\Bold{F}_{#1}}'
```

If the Sage object is not in the global name space, describe it like so:

```
sage: produce_latex_macro('sage.rings.finite_rings.constructor.FiniteField', 3)
'\\newcommand{\\FiniteField}[1]{\\Bold{F}_{#1}}'
```

`sage.misc.latex_macros.`**`sage_latex_macros`**`()`

Return list of LaTeX macros for Sage. This just runs the function `produce_latex_macro()` on the list `macros` defined in this file, and appends `sage_configurable_latex_macros`. To add a new macro for permanent use in Sage, modify `macros`.

EXAMPLES:

```
sage: from sage.misc.latex_macros import sage_latex_macros
sage: sage_latex_macros()
['\\newcommand{\\ZZ}{\\Bold{Z}}', '\\newcommand{\\NN}{\\Bold{N}}', ...
```

`sage.misc.latex_macros.`**`sage_mathjax_macros`**`()`

Return list of MathJax macro definitions for Sage as JavaScript. This feeds each item output by `sage_latex_macros()` to `convert_latex_macro_to_mathjax()`.

EXAMPLES:

```
sage: from sage.misc.latex_macros import sage_mathjax_macros
sage: sage_mathjax_macros()
['ZZ: "\\\\Bold{Z}"', 'NN: "\\\\Bold{N}"', ...
```

CHAPTER

FIFTYONE

LAZYNESS

51.1 Lazy attributes

AUTHORS:

- Nicolas Thiery (2008): Initial version
- Nils Bruin (2013-05): Cython version

class `sage.misc.lazy_attribute.`**`lazy_attribute`**`(f)`

Bases: `sage.misc.lazy_attribute._lazy_attribute`

A lazy attribute for an object is like a usual attribute, except that, instead of being computed when the object is constructed (i.e. in `__init__`), it is computed on the fly the first time it is accessed.

For constant values attached to an object, lazy attributes provide a shorter syntax and automatic caching (unlike methods), while playing well with inheritance (like methods): a subclass can easily override a given attribute; you don't need to call the super class constructor, etc.

Technically, a `lazy_attribute` is a non-data descriptor (see Invoking Descriptors in the Python reference manual).

EXAMPLES:

We create a class whose instances have a lazy attribute `x`:

```
sage: class A(object):
....:     def __init__(self):
....:         self.a=2 # just to have some data to calculate from
....:
....:     @lazy_attribute
....:     def x(self):
....:         print "calculating x in A"
....:         return self.a + 1
....:
```

For an instance `a` of `A`, `a.x` is calculated the first time it is accessed, and then stored as a usual attribute:

```
sage: a = A()
sage: a.x
calculating x in A
3
sage: a.x
3
```

Implementation details

We redo the same example, but opening the hood to see what happens to the internal dictionary of the object:

```
sage: a = A()
sage: a.__dict__
{'a': 2}
sage: a.x
calculating x in A
3
sage: a.__dict__
{'a': 2, 'x': 3}
sage: a.x
3
sage: timeit('a.x') # random
625 loops, best of 3: 89.6 ns per loop
```

This shows that, after the first calculation, the attribute `x` becomes a usual attribute; in particular, there is no time penalty to access it.

A lazy attribute may be set as usual, even before its first access, in which case the lazy calculation is completely ignored:

```
sage: a = A()
sage: a.x = 4
sage: a.x
4
sage: a.__dict__
{'a': 2, 'x': 4}
```

Class binding results in the lazy attribute itself:

```
sage: A.x
<sage.misc.lazy_attribute.lazy_attribute object at ...>
```

Conditional definitions

The function calculating the attribute may return NotImplemented to declare that, after all, it is not able to do it. In that case, the attribute lookup proceeds in the super class hierarchy:

```
sage: class B(A):
....:     @lazy_attribute
....:     def x(self):
....:         if hasattr(self, "y"):
....:             print "calculating x from y in B"
....:             return self.y
....:         else:
....:             print "y not there; B does not define x"
....:             return NotImplemented
....:
sage: b = B()
sage: b.x
y not there; B does not define x
calculating x in A
3
sage: b = B()
sage: b.y = 1
sage: b.x
```

```
calculating x from y in B
1
```

Attribute existence testing

Testing for the existence of an attribute with hasattr currently always triggers its full calculation, which may not be desirable when the calculation is expensive:

```
sage: a = A()
sage: hasattr(a, "x")
calculating x in A
True
```

It would be great if we could take over the control somehow, if at all possible without a special implementation of hasattr, so as to allow for something like:

```
sage: class A (object):
....:     @lazy_attribute
....:     def x(self, existence_only=False):
....:         if existence_only:
....:             print "testing for x existence"
....:             return True
....:         else:
....:             print "calculating x in A"
....:             return 3
....:
sage: a = A()
sage: hasattr(a, "x") # todo: not implemented
testing for x existence
sage: a.x
calculating x in A
3
sage: a.x
3
```

Here is a full featured example, with both conditional definition and existence testing:

```
sage: class B(A):
....:     @lazy_attribute
....:     def x(self, existence_only=False):
....:         if hasattr(self, "y"):
....:             if existence_only:
....:                 print "testing for x existence in B"
....:                 return True
....:             else:
....:                 print "calculating x from y in B"
....:                 return self.y
....:         else:
....:             print "y not there; B does not define x"
....:             return NotImplemented
....:
sage: b = B()
sage: hasattr(b, "x") # todo: not implemented
y not there; B does not define x
testing for x existence
True
sage: b.x
y not there; B does not define x
```

```
calculating x in A
3
sage: b = B()
sage: b.y = 1
sage: hasattr(b, "x") # todo: not implemented
testing for x existence in B
True
sage: b.x
calculating x from y in B
1
```

lazy attributes and introspection

Todo

Make the following work nicely:

```
sage: b.x?                # todo: not implemented
sage: b.x??               # todo: not implemented
```

Right now, the first one includes the doc of this class, and the second one brings up the code of this class, both being not very useful.

TESTS:

Partial support for old style classes

Old style and new style classes play a bit differently with @property and attribute setting:

```
sage: class A:
....:     @property
....:     def x(self):
....:         print "calculating x"
....:         return 3
....:
sage: a = A()
sage: a.x = 4
sage: a.__dict__
{'x': 4}
sage: a.x
4
sage: a.__dict__['x']=5
sage: a.x
5

sage: class A (object):
....:     @property
....:     def x(self):
....:         print "calculating x"
....:         return 3
....:
sage: a = A()
sage: a.x = 4
Traceback (most recent call last):
...
```

```
AttributeError: can't set attribute
sage: a.__dict__
{}
sage: a.x
calculating x
3
sage: a.__dict__['x']=5
sage: a.x
calculating x
3
```

In particular, lazy_attributes need to be implemented as non-data descriptors for new style classes, so as to leave access to setattr. We now check that this implementation also works for old style classes (conditional definition does not work yet):

```
sage: class A:
....:     def __init__(self):
....:         self.a=2 # just to have some data to calculate from
....:
....:     @lazy_attribute
....:     def x(self):
....:         print "calculating x"
....:         return self.a + 1
....:
sage: a = A()
sage: a.__dict__
{'a': 2}
sage: a.x
calculating x
3
sage: a.__dict__
{'a': 2, 'x': 3}
sage: a.x
3
sage: timeit('a.x') # random
625 loops, best of 3: 115 ns per loop

sage: a = A()
sage: a.x = 4
sage: a.x
4
sage: a.__dict__
{'a': 2, 'x': 4}

sage: class B(A):
....:     @lazy_attribute
....:     def x(self):
....:         if hasattr(self, "y"):
....:             print "calculating x from y in B"
....:             return self.y
....:         else:
....:             print "y not there; B does not define x"
....:             return NotImplemented
....:
sage: b = B()
sage: b.x                          # todo: not implemented
y not there; B does not define x
calculating x in A
```

```
3
sage: b = B()
sage: b.y = 1
sage: b.x
calculating x from y in B
1
```

Lazy attributes and Cython

This attempts to check that lazy attributes work with built-in functions like cpdef methods:

```
sage: class A:
....:     def __len__(x):
....:         return int(5)
....:     len = lazy_attribute(len)
....:
sage: A().len
5
```

Since trac ticket #11115, extension classes derived from `Parent` can inherit a lazy attribute, such as `element_class`:

```
sage: cython_code = ["from sage.structure.parent cimport Parent",
....: "from sage.structure.element cimport Element",
....: "cdef class MyElement(Element): pass",
....: "cdef class MyParent(Parent):",
....: "    Element = MyElement"]
sage: cython('\n'.join(cython_code))
sage: P = MyParent(category=Rings())
sage: P.element_class    # indirect doctest
<type '...MyElement'>
```

About descriptor specifications

The specifications of descriptors (see 3.4.2.3 Invoking Descriptors in the Python reference manual) are incomplete w.r.t. inheritance, and maybe even ill-implemented. We illustrate this on a simple class hierarchy, with an instrumented descriptor:

```
sage: class descriptor(object):
....:     def __get__(self, obj, cls):
....:         print cls
....:         return 1
sage: class A(object):
....:     x = descriptor()
sage: class B(A):
....:     pass
....:
```

This is fine:

```
sage: A.x
<class '__main__.A'>
1
```

The behaviour for the following case is not specified (see Instance Binding) when `x` is not in the dictionary of `B` but in that of some super category:

```
sage: B().x
<class '__main__.B'>
1
```

It would seem more natural (and practical!) to get `A` rather than `B`.

From the specifications for Super Binding, it would be expected to get `A` and not `B` as cls parameter:

```
sage: super(B, B()).x
<class '__main__.B'>
1
```

Due to this, the natural implementation runs into an infinite loop in the following example:

```
sage: class A(object):
....:     @lazy_attribute
....:     def unimplemented_A(self):
....:         return NotImplemented
....:     @lazy_attribute
....:     def unimplemented_AB(self):
....:         return NotImplemented
....:     @lazy_attribute
....:     def unimplemented_B_implemented_A(self):
....:         return 1
....:
sage: class B(A):
....:     @lazy_attribute
....:     def unimplemented_B(self):
....:         return NotImplemented
....:     @lazy_attribute
....:     def unimplemented_AB(self):
....:         return NotImplemented
....:     @lazy_attribute
....:     def unimplemented_B_implemented_A(self):
....:         return NotImplemented
....:
sage: class C(B):
....:     pass
....:
```

This is the simplest case where, without workaround, we get an infinite loop:

```
sage: hasattr(B(), "unimplemented_A") # todo: not implemented
False
```

Todo

Improve the error message:

```
sage: B().unimplemented_A # todo: not implemented
Traceback (most recent call last):
...
AttributeError: 'super' object has no attribute 'unimplemented_A'
```

We now make some systematic checks:

```
sage: B().unimplemented_A
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_A'
```

```
sage: B().unimplemented_B
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_B'
sage: B().unimplemented_AB
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_AB'
sage: B().unimplemented_B_implemented_A
1

sage: C().unimplemented_A()
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_A'
sage: C().unimplemented_B()
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_B'
sage: C().unimplemented_AB()
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_AB'
sage: C().unimplemented_B_implemented_A # todo: not implemented
1
```

class sage.misc.lazy_attribute.**lazy_class_attribute**(*f*)

Bases: sage.misc.lazy_attribute.lazy_attribute

A lazy class attribute for an class is like a usual class attribute, except that, instead of being computed when the class is constructed, it is computed on the fly the first time it is accessed, either through the class itself or trough on of its objects.

This is very similar to lazy_attribute except that the attribute is a class attribute. More precisely, once computed, the lazy class attribute is stored in the class rather than in the object. The lazy class attribute is only computed once for all the objects:

```
sage: class Cl(object):
....:     @lazy_class_attribute
....:     def x(cls):
....:          print "computing x"
....:          return 1
sage: Cl.x
computing x
1
sage: Cl.x
1
```

As for a any usual class attribute it is also possible to access it from an object:

```
sage: b = Cl()
sage: b.x
1
```

First access from an object also porperly triggers the computation:

```
sage: class Cl1(object):
....:     @lazy_class_attribute
....:     def x(cls):
....:          print "computing x"
```

```
....:         return 1
sage: Cl1().x
computing x
1
sage: Cl1().x
1
```

..WARNING:

```
The behavior of lazy class attributes with respect to inheritance is
not specified. It currently depends on the evaluation order::

    sage: class A(object):
    ....:     @lazy_class_attribute
    ....:     def x(cls):
    ....:          print "computing x"
    ....:          return str(cls)
    ....:     @lazy_class_attribute
    ....:     def y(cls):
    ....:          print "computing y"
    ....:          return str(cls)
    sage: class B(A):
    ....:     pass

    sage: A.x
    computing x
    "<class '__main__.A'>"
    sage: B.x
    "<class '__main__.A'>"

    sage: B.y
    computing y
    "<class '__main__.B'>"
    sage: A.y
    computing y
    "<class '__main__.A'>"
    sage: B.y
    "<class '__main__.B'>"
```

TESTS:

```
sage: "x" in b.__dict__
False
```

51.2 Lazy format strings

class `sage.misc.lazy_format.`**`LazyFormat`**

Bases: `str`

Lazy format strings

An instance of `LazyFormat` behaves like a usual format string, except that the evaluation of the `__repr__` method of the formated arguments it postponed until actual printing.

EXAMPLES:

Under normal circumstances, `Lazyformat` strings behave as usual:

```
sage: from sage.misc.lazy_format import LazyFormat
sage: LazyFormat("Got `%s`; expected a list")%3
Got `3`; expected a list
sage: LazyFormat("Got `%s`; expected %s")%(3, 2/3)
Got `3`; expected 2/3
```

To demonstrate the lazyness, let us build an object with a broken `__repr__` method:

```
sage: class IDontLikeBeingPrinted(object):
...     def __repr__(self):
...         raise ValueError("Don't ever try to print me !")
```

There is no error when binding a lazy format with the broken object:

```
sage: lf = LazyFormat("<%s>")%IDontLikeBeingPrinted()
```

The error only occurs upon printing:

```
sage: lf
<repr(<sage.misc.lazy_format.LazyFormat at 0x...>) failed: ValueError: Don't ever try to print m
```

Common use case:

Most of the time, `__repr__` methods are only called during user interaction, and therefore need not be fast; and indeed there are objects `x` in Sage such `x.__repr__()` is time consuming.

There are however some uses cases where many format strings are constructed but not actually printed. This includes error handling messages in `unittest` or `TestSuite` executions:

```
sage: QQ._tester().assertTrue(0 in QQ,
...                    "%s doesn't contain 0"%QQ)
```

In the above `QQ.__repr__()` has been called, and the result immediately discarded. To demonstrate this we replace `QQ` in the format string argument with our broken object:

```
sage: QQ._tester().assertTrue(True,
...                    "%s doesn't contain 0"%IDontLikeBeingPrinted())
Traceback (most recent call last):
...
ValueError: Don't ever try to print me !
```

This behavior can induce major performance penalties when testing. Note that this issue does not impact the usual assert:

```
sage: assert True, "%s is wrong"%IDontLikeBeingPrinted()
```

We now check that `LazyFormat` indeed solves the assertion problem:

```
sage: QQ._tester().assertTrue(True,
...                   LazyFormat("%s is wrong")%IDontLikeBeingPrinted())
sage: QQ._tester().assertTrue(False,
...                   LazyFormat("%s is wrong")%IDontLikeBeingPrinted())
Traceback (most recent call last):
...
AssertionError: <unprintable AssertionError object>
```

51.3 Lazy imports

This module allows one to lazily import objects into a namespace, where the actual import is delayed until the object is actually called or inspected. This is useful for modules that are expensive to import or may cause circular references, though there is some overhead in its use.

EXAMPLES:

```
sage: from sage.misc.lazy_import import lazy_import
sage: lazy_import('sage.rings.all', 'ZZ')
sage: type(ZZ)
<type 'sage.misc.lazy_import.LazyImport'>
sage: ZZ(4.0)
4
```

By default, a warning is issued if a lazy import module is resolved during Sage's startup. In case a lazy import's sole purpose is to break a circular reference and it is known to be resolved at startup time, one can use the `at_startup` option:

```
sage: lazy_import('sage.rings.all', 'ZZ', at_startup=True)
```

This option can also be used as an intermediate step toward not importing by default a module that is used in several places, some of which can already afford to lazy import the module but not all.

A lazy import that is marked as "at_startup" will print a message if it is actually resolved after the startup, so that the developer knows that (s)he can remove the flag:

```
sage: ZZ
Option ``at_startup=True`` for lazy import ZZ not needed anymore
Integer Ring
```

See also:

`lazy_import()`, `LazyImport`

AUTHOR:

- Robert Bradshaw

class `sage.misc.lazy_import.`**`LazyImport`**

Bases: `object`

EXAMPLES:

```
sage: from sage.misc.lazy_import import LazyImport
sage: my_integer = LazyImport('sage.rings.all', 'Integer')
sage: my_integer(4)
4
sage: my_integer('101', base=2)
5
sage: my_integer(3/2)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

`sage.misc.lazy_import.`**`finish_startup`**`()`

This function must be called exactly once at the end of the Sage import process

TESTS:

```
sage: from sage.misc.lazy_import import finish_startup
sage: finish_startup()
Traceback (most recent call last):
...
AssertionError: finish_startup() must be called exactly once
```

sage.misc.lazy_import.**get_star_imports**(*module_name*)

Lookup the list of names in a module that would be imported with "import *" either via a cache or actually importing.

EXAMPLES:

```
sage: from sage.misc.lazy_import import get_star_imports
sage: 'get_star_imports' in get_star_imports('sage.misc.lazy_import')
True
sage: 'EllipticCurve' in get_star_imports('sage.schemes.all')
True
```

TESTS:

```
sage: import os, tempfile
sage: fd, cache_file = tempfile.mkstemp()
sage: os.write(fd, 'invalid')
7
sage: os.close(fd)
sage: import sage.misc.lazy_import as lazy
sage: lazy.get_cache_file = (lambda: cache_file)
sage: lazy.star_imports = None
sage: lazy.get_star_imports('sage.schemes.all')
doctest:...: UserWarning: star_imports cache is corrupted
[...]
sage: os.remove(cache_file)
```

sage.misc.lazy_import.**is_during_startup**()

Return whether Sage is currently starting up

OUTPUT:

Boolean

TESTS:

```
sage: from sage.misc.lazy_import import is_during_startup
sage: is_during_startup()
False
```

sage.misc.lazy_import.**lazy_import**(*module*, *names*, *_as=None*, *namespace=None*, *overwrite=True*, *at_startup=False*, *deprecation=None*)

Create a lazy import object and inject it into the caller's global namespace. For the purposes of introspection and calling, this is like performing a lazy "from module import name" where the import is delayed until the object actually is used or inspected.

INPUT:

- `module` – a string representing the module to import
- `names` – a string or list of strings representing the names to import from module
- `_as` – (optional) a string or list of strings representing the aliases of the names imported
- `namespace` – the namespace where importing the names; by default, import the names to current namespace

- `overwrite` – (default: `True`) if set to `True` and a name is already in the namespace, overwrite it with the lazy_import-ed name
- `at_startup` – a boolean (default: `False`); whether the lazy import is supposed to be resolved at startup time
- `deprecation` – (optional) if not `None`, a deprecation warning will be issued when the object is actually imported; `deprecation` should be either a trac number (integer) or a pair `(trac_number, message)`

See also:

`sage.misc.lazy_import`, `LazyImport`

EXAMPLES:

```
sage: from sage.misc.lazy_import import lazy_import
sage: lazy_import('sage.rings.all', 'ZZ')
sage: type(ZZ)
<type 'sage.misc.lazy_import.LazyImport'>
sage: ZZ(4.0)
4
sage: lazy_import('sage.rings.all', 'RDF', 'my_RDF')
sage: my_RDF._get_object() is RDF
True
sage: my_RDF(1/2)
0.5

sage: lazy_import('sage.all', ['QQ', 'RR'], ['my_QQ', 'my_RR'])
sage: my_QQ._get_object() is QQ
True
sage: my_RR._get_object() is RR
True
```

Upon the first use, the object is injected directly into the calling namespace:

```
sage: lazy_import('sage.all', 'ZZ', 'my_ZZ')
sage: my_ZZ is ZZ
False
sage: my_ZZ(37)
37
sage: my_ZZ is ZZ
True
```

We check that `lazy_import()` also works for methods:

```
sage: class Foo(object):
...       lazy_import('sage.all', 'plot')
sage: class Bar(Foo):
...       pass
sage: type(Foo.__dict__['plot'])
<type 'sage.misc.lazy_import.LazyImport'>
sage: 'EXAMPLES' in Bar.plot.__doc__
True
sage: type(Foo.__dict__['plot'])
<type 'function'>
```

If deprecated then a deprecation warning is issued:

```
sage: lazy_import('sage.all', 'Qp', 'my_Qp', deprecation=14275)
sage: my_Qp(5)
doctest:...: DeprecationWarning:
```

```
Importing my_Qp from here is deprecated. If you need to use it, please import it directly from s
See http://trac.sagemath.org/14275 for details.
5-adic Field with capped relative precision 20
```

An example of deprecation with a message:

```
sage: lazy_import('sage.all', 'Qp', 'my_Qp_msg', deprecation=(14275, "This is an example."))
sage: my_Qp_msg(5)
doctest:...: DeprecationWarning: This is an example.
See http://trac.sagemath.org/14275 for details.
5-adic Field with capped relative precision 20
```

sage.misc.lazy_import.**save_cache_file**()

Used to save the cached * import names.

TESTS:

```
sage: import sage.misc.lazy_import
sage: sage.misc.lazy_import.save_cache_file()
```

sage.misc.lazy_import.**test_fake_startup**()

For testing purposes only.

Switch the startup lazy import guard back on.

EXAMPLES:

```
sage: sage.misc.lazy_import.test_fake_startup()
sage: from sage.misc.lazy_import import lazy_import
sage: lazy_import('sage.rings.all', 'ZZ', 'my_ZZ')
sage: my_ZZ(123)
------------------------------------------------------------------------------
Resolving lazy import ZZ during startup
Calling stack:
...
------------------------------------------------------------------------------
123
sage: sage.misc.lazy_import.finish_startup()
```

CHAPTER FIFTYTWO

INDICES AND TABLES

- Index
- Module Index
- Search Page

BIBLIOGRAPHY

[Apostol] T. Apostol, Modular functions and Dirichlet series in number theory, Springer, 1997 (2nd ed), section 3.7–3.9.

[UNITTEST] unittest – Unit testing framework – http://docs.python.org/library/unittest.html

m

r

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z