## CONTENTS

1 General Infrastructure .................................................. 1
2 Programming Utilities .................................................. 91
3 Code Evaluation .......................................................... 357
4 Formatted Output ....................................................... 385
5 Saving and Loading Sage Objects ................................. 519
6 Interactive Usage Support ............................................. 569
7 Development Tools ..................................................... 607
8 Low-Level Utilities ...................................................... 689
9 Indices and Tables ....................................................... 719

Python Module Index ...................................................... 721
Index .............................................................................. 723
1.1 Default Settings

AUTHORS: William Stein and David Kohel

`sage.misc.defaults.latex_variable_names(n, name=None)`

Convert a root string into a tuple of variable names by adding numbers in sequence.

**INPUT:**

- `n` a non-negative Integer; the number of variable names to output
- `names` a string (default: `None`); the root of the variable name.

**EXAMPLES:**

```python
sage: from sage.misc.defaults import latex_variable_names
sage: latex_variable_names(0)
()  
sage: latex_variable_names(1, 'a')
('a',)  
sage: latex_variable_names(3, beta)
('beta_0', 'beta_1', 'beta_2')  
sage: latex_variable_names(3, r'\beta')
('\beta_0', '\beta_1', '\beta_2')
```

```python
>>> from sage.all import *
>>> from sage.misc.defaults import latex_variable_names

>>> latex_variable_names(Integer(0))
()  
>>> latex_variable_names(Integer(1), 'a')
('a',)  
>>> latex_variable_names(Integer(3), beta)
('beta_0', 'beta_1', 'beta_2')  
>>> latex_variable_names(Integer(3), r'\beta')
('\beta_0', '\beta_1', '\beta_2')
```

`sage.misc.defaults.series_precision()`

Return the Sage-wide precision for series (symbolic, power series, Laurent series).

**EXAMPLES:**

```python
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)

Convert a root string into a tuple of variable names by adding numbers in sequence.

INPUT:

- `n` a non-negative Integer; the number of variable names to output
- `name` a string (default: None); the root of the variable name.

EXAMPLES:

```
sage: from sage.misc.defaults import variable_names
sage: variable_names(0)
()
sage: variable_names(1)
('x',)
sage: variable_names(1, 'alpha')
('alpha',)
sage: variable_names(2, 'alpha')
('alpha0', 'alpha1')
```
1.2 Functional notation

These are functions so that you can write $\text{foo}(x)$ instead of $x.\text{foo}()$ 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)
```

Return a numerical approximation of self with prec bits (or decimal digits) of precision.

No guarantee is made about the accuracy of the result.

**Note:** Lower case $n()$ is an alias for `numerical_approx()` and may be used as a method.

**INPUT:**
- prec – precision in bits
- digits – precision in decimal digits (only used if prec is not given)
- algorithm – which algorithm to use to compute this approximation (the accepted algorithms depend on the object)

If neither prec nor digits is given, the default precision is 53 bits (roughly 16 digits).

**EXAMPLES:**

```
sage: # needs sage.symbolic
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: # needs sage.rings.real_mpfr
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
```

```python
>>> from sage.all import *
>>> # needs sage.symbolic
>>> numerical_approx(pi, Integer(10))
3.1
```

(continues on next page)
You can also usually use method notation:

\[
\begin{align*}
\text{sage: } & (\pi^2 + e).n() & \quad \# \text{ needs sage.symbolic} \\
& 12.5878862295484 & \\
\text{sage: } & (\pi^2 + e).\text{numerical\_approx}() & \quad \# \text{ needs sage.symbolic} \\
& 12.5878862295484 & \\
\end{align*}
\]

Vectors and matrices may also have their entries approximated:

\[
\begin{align*}
\text{sage: } & v = \text{vector}(\text{RDF}, [1, 2, 3]) & \quad \# \text{ needs sage.modules} \\
& \text{v.n()} & \quad \# \text{ needs sage.modules} \\
& (1.00000000000000, 2.00000000000000, 3.00000000000000) & \\
\text{sage: } & \text{v.n(prec=20)} & \quad \# \text{ needs sage.modules} \\
& \text{v.n(prec=20)} & \\
& (1.00000000000000, 2.00000000000000, 3.00000000000000) & \\
\text{sage: } & \text{v.parent()} & \\
& \text{Vector space of dimension 3 over Complex Field with 53 bits of precision} & \\
\text{sage: } & u = \text{vector}(\text{QQ}, [1/2, 1/3, 1/4]) & \quad \# \text{ needs sage.modules} \\
\end{align*}
\]
needs sage.modules

sage: n(u, prec=15)  
# needs sage.modules
(0.5000, 0.3333, 0.2500)

sage: n(u, digits=5)  
# needs sage.modules
(0.50000, 0.33333, 0.25000)

# needs sage.modules

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)

# needs sage.modules

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, range(24))

sage: N(B, digits=2)

\[0.00 1.0 2.0 3.0 4.0 5.0 6.0 7.0
8.0 9.0 10. 11. 0.00 1.0 2.0 3.0
4.0 5.0 6.0 7.0 8.0 9.0 10. 11.\]

>>> from sage.all import *

>>> v = vector(RDF, [Integer(1),Integer(2),Integer(3)])  
# needs sage.modules

>>> v.n()  
# needs sage.modules
(1.00000000000000, 2.00000000000000, 3.00000000000000)

>>> v = vector(CDF, [Integer(1),Integer(2),Integer(3)])

>>> v.n()  
# needs sage.modules
(1.00000000000000, 2.00000000000000, 3.00000000000000)

>>> _.parent()  
Vector space of dimension 3 over Complex Field with 53 bits of precision

>>> v.n(prec=Integer(20))

(1.0000, 2.0000, 3.0000)

>>> u = vector(QQ, [Integer(1)/Integer(2), Integer(1)/Integer(3), Integer(1)/
Integer(4)], sparse=True)  
# needs sage.modules

>>> n(u, prec=Integer(15))  
# needs sage.modules
(0.5000, 0.3333, 0.2500)

>>> n(u, digits=Integer(5))  
# needs sage.modules
(0.50000, 0.33333, 0.25000)

>>> v = vector(QQ, [Integer(1)/Integer(2), Integer(0), Integer(0), Integer(1)/
Integer(3), Integer(0), Integer(0), Integer(0), Integer(1)/Integer(4)],
sparse=True)  
# needs sage.modules
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  # needs sage.symbolic
3.14
sage: y = N(3.14, digits=3); y  # needs sage.rings.real_mpfr
3.14
sage: x == y  # needs sage.rings.real_mpfr sage.symbolic
False
sage: x.str(base=2)  # needs sage.symbolic
'11.001001000100'
sage: y.str(base=2)  # needs sage.rings.real_mpfr
'11.001000111101'
```

Increasing the precision of a floating point number is not allowed:

```
sage: CC(-5).n(prec=100)
```

(continues on next page)
As an exceptional case, digits=1 usually leads to 2 digits (one significant) in the decimal output (see Issue #11647):

```
sage: # needs sage.symbolic
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:

```python
sage: N(pi, prec=2)  # needs sage.symbolic
3.0
sage: N(3, prec=2)   # needs sage.rings.real_mpfr
3.0
sage: N(pi - 3, prec=2)  # needs sage.symbolic
0.00
```

```python
>>> from sage.all import *
>>> N(pi, prec=Integer(2))  # needs sage.symbolic
>>> N(Integer(3), prec=Integer(2))  # needs sage.rings.real_mpfr
>>> N(pi - Integer(3), prec=Integer(2))  # needs sage.symbolic
0.00
```

sage.misc.functional.additive_order(x)

Return the additive order of x.

EXAMPLES:

```python
sage: additive_order(5)
+Infinity
sage: additive_order(Mod(5,11))
11
sage: additive_order(Mod(4,12))
3
```

```python
>>> from sage.all import *
>>> additive_order(Integer(5))
>>> additive_order(Mod(Integer(5),Integer(11)))
>>> additive_order(Mod(Integer(4),Integer(12)))
```

sage.misc.functional.base_field(x)

Return the base field over which x is defined.

EXAMPLES:

```python
sage: R = PolynomialRing(GF(7), 'x')
sage: base_ring(R)
Finite Field of size 7
sage: base_field(R)
Finite Field of size 7
```

```python
>>> from sage.all import *
>>> R = PolynomialRing(GF(Integer(7)), 'x')
>>> base_ring(R)
(continues on next page)```
Finite Field of size 7
>>> 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:

```python
sage: R.base_field()
Traceback (most recent call last):
  ...  
AttributeError: 'PolynomialRing_dense_mod_p_with_category' object has no_
→attribute 'base_field'...
```

```python
>>> from sage.all import *
```  
```python
>>> 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)`

Return the base ring over which `x` is defined.

**EXAMPLES:**

```python
sage: R = PolynomialRing(GF(7), 'x')
```

```python
sage: base_ring(R)
```

```python
Finite Field of size 7
```

```python
>>> from sage.all import *
```  
```python
>>> R = PolynomialRing(GF(Integer(7)), 'x')
>>> base_ring(R)
```

```python
Finite Field of size 7
```

`sage.misc.functional.basis(x)`

Return the fixed basis of `x`.

**EXAMPLES:**

```python
sage: V = VectorSpace(QQ, 3)  # needs sage.modules
```

```python
sage: S = V.subspace([[1,2,0], [2,2,-1]])  # needs sage.modules
```

```python
sage: basis(S)  # needs sage.modules
```

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

```python
>>> from sage.all import *
```  
```python
>>> V = VectorSpace(QQ, Integer(3))  # needs sage.modules
```

```python
>>> S = V.subspace([[Integer(1),Integer(2),Integer(0)], [Integer(2),Integer(2),-
Integer(1)]]])  # needs sage.modules
```

(continues on next page)

1.2. Functional notation 9
sage.misc.functional.category(x)

Return the category of x.

EXAMPLES:

```python
sage: V = VectorSpace(QQ, 3)
# needs sage.modules
sage: category(V)
# needs sage.modules
Category of finite dimensional vector spaces with basis over
(number fields and quotient fields and metric spaces)
```

sage.misc.functional.characteristic_polynomial(x, var='x')

Return the characteristic polynomial of x in the given variable.

EXAMPLES:

```python
sage: # needs sage.libs.pari sage.modules
sage: M = MatrixSpace(QQ, 3, 3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: charpoly(A)
```

(continues on next page)
x^3 - 15*x^2 - 18*x
>>> charpoly(A, 't')
t^3 - 15*t^2 - 18*t

>>> k = GF(Integer(7)**Integer(10), names=('alpha',)); (alpha,) = k._first_
˓→ngens(1); k  # needs sage.
Finite Field in alpha of size 7^10
>>> alpha.charpoly('T') # needs sage.rings.finite_rings
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 do not confuse the computation (Issue #14403):

Ensure the variable name of the polynomial does not conflict with variables used within the matrix, and that non-integral powers of variables do not confuse the computation (Issue #14403):

sage: y = var('y')
sage: a = 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')

EXAMPLES:
Finite Field in alpha of size 7^10
sage: alpha.charpoly('T')  # needs sage.rings.finite_rings
T^10 + T^6 + T^5 + 4*T^4 + T^3 + 2*T^2 + 3*T + 3
sage: characteristic_polynomial(alpha, 'T')  # needs sage.rings.finite_rings
T^10 + T^6 + T^5 + 4*T^4 + T^3 + 2*T^2 + 3*T + 3

>>> from sage.all import *  # needs sage.libs.pari sage.modules
>>> M = MatrixSpace(QQ, Integer(3), Integer(3))
>>> A = M([[Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),Integer(6)],
        [Integer(7),Integer(8),Integer(9)])
>>> charpoly(A)
x^3 - 15*x^2 - 18*x
>>> charpoly(A, 't')
t^3 - 15*t^2 - 18*t

>>> k = GF(Integer(7)**Integer(10), names=('alpha',)); (alpha,) = k._first_ngens(1); k  # needs sage.rings.finite_rings
Finite Field in alpha of size 7^10
>>> alpha.charpoly('T')  # needs sage.rings.finite_rings
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 do not confuse the computation (Issue #14403):

sage: # needs sage.libs.pari sage.symbolic
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]

>>> from sage.all import *  # needs sage.libs.pari sage.symbolic
>>> y = var('y')
>>> a = matrix([[x,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]])
>>> characteristic_polynomial(a).list()
x, -3*x - 1, 3*x + 3, -x - 3, 1
>>> b = matrix([[y**(1/2),0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]])
>>> charpoly(b).list()
[sqrt(y), -3*sqrt(y) - 1, 3*sqrt(y) + 3, -sqrt(y) - 3, 1]

sage.misc.functional.coerce(P,x)
Coerce $x$ to type $\mathbb{F}$ if possible.

**EXAMPLES:**

```
sage: type(5)
<class 'sage.rings.integer.Integer'>
sage: type(coerce(QQ,5))
<class 'sage.rings.rational.Rational'>
```

```
>>> from sage.all import *
>>> type(Integer(5))
<class 'sage.rings.integer.Integer'>
>>> type(coerce(QQ,Integer(5)))
<class 'sage.rings.rational.Rational'>
```

`sage.misc.functional.cyclotomic_polynomial(n, var='x')`

Return the $n^{th}$ cyclotomic polynomial.

**EXAMPLES:**

```
sage: # needs sage.libs.pari
cyclotomic_polynomial(3)
x^2 + x + 1
cyclotomic_polynomial(4)
x^2 + 1
cyclotomic_polynomial(9)
x^6 + x^3 + 1
cyclotomic_polynomial(10)
x^4 - x^3 + x^2 - x + 1
cyclotomic_polynomial(11)
x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
```

```
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> cyclotomic_polynomial(Integer(3))
x^2 + x + 1
>>> cyclotomic_polynomial(Integer(4))
x^2 + 1
>>> cyclotomic_polynomial(Integer(9))
x^6 + x^3 + 1
>>> cyclotomic_polynomial(Integer(10))
x^4 - x^3 + x^2 - x + 1
>>> cyclotomic_polynomial(Integer(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)`

Return the decomposition of $x$.

**EXAMPLES:**

```
sage: M = matrix([[2, 3], [3, 4]])
# needs sage.libs.pari sage.modules
sage: M.decomposition()
# ...
(Ambient free module of rank 2 over the principal ideal domain Integer Ring, True)
```

(continues on next page)
Utilities, Release 10.4

```python
sage: # needs sage.groups
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 modulo 4
with values in Cyclotomic Field of order 4 and degree 2
```

```python
>>> from sage.all import *
>>> M = matrix([[Integer(2), Integer(3)], [Integer(3), Integer(4)]]);
# needs sage.libs.pari sage.modules
>>> M.decomposition()
# needs sage.libs.pari sage.modules
[ (Ambient free module of rank 2 over the principal ideal domain Integer Ring, True) ]
```

```python
sage.misc.functional.denominator(x)

Return the denominator of x.

EXAMPLES:

```python
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: M = MatrixSpace(QQ, 3, 3)  # needs sage.modules
          
M = MatrixSpace(QQ, 3, 3)
          
sage: A = M([1,2,3, 4,5,6, 7,8,9])  # needs sage.modules
          
A = MatrixSpace(QQ, 3, 3)
          
sage: det(A)  # needs sage.modules
0
          
>>> from sage.all import *
    
M = MatrixSpace(QQ, Integer(3), Integer(3))  # needs sage.modules
    
A = MatrixSpace(QQ, Integer(3), Integer(3))
    
>>> A = M([Integer(1),Integer(2),Integer(3), Integer(4),Integer(5),Integer(6),Integer(7),Integer(8),Integer(9)])
    # needs sage.modules
    
>>> det(A)  # needs sage.modules
0

sage.misc.functional.dim(x)

Return the dimension of x.

EXAMPLES:

sage: V = VectorSpace(QQ, 3)  # needs sage.modules
          
V = VectorSpace(QQ, 3)
          
sage: S = V.subspace([[1,2,0], [2,2,-1]])  # needs sage.modules
          
S = V.subspace([[1,2,0], [2,2,-1]])
          
sage: dimension(S)  # needs sage.modules
2
          
>>> from sage.all import *
    
V = VectorSpace(QQ, Integer(3))  # needs sage.modules
    
V = VectorSpace(QQ, Integer(3))
    
>>> S = V.subspace([[Integer(1),Integer(2),Integer(0)], [Integer(2),Integer(2),-Integer(1)]])  # needs sage.modules
    
S = V.subspace([[Integer(1),Integer(2),Integer(0)], [Integer(2),Integer(2),-Integer(1)]])
    
>>> dimension(S)  # needs sage.modules
2

sage.misc.functional.dimension(x)

Return the dimension of x.

EXAMPLES:

sage: V = VectorSpace(QQ, 3)  # needs sage.modules
          
V = VectorSpace(QQ, 3)
          
sage: S = V.subspace([[1,2,0], [2,2,-1]])  # needs sage.modules
          
S = V.subspace([[1,2,0], [2,2,-1]])
          
sage: dimension(S)  # needs sage.modules
2
          
>>> from sage.all import *
    
V = VectorSpace(QQ, Integer(3))  # needs sage.modules
    
V = VectorSpace(QQ, Integer(3))
    
(continues on next page)
Utilities, Release 10.4

(continued from previous page)

```python
>>> S = V.subspace([[Integer(1),Integer(2),Integer(0)], [Integer(2),Integer(2),-
                 Integer(1)]])          # needs sage.modules
>>> dimension(S)                   #...
    # needs sage.modules
2
```

`sage.misc.functional.disc(x)`

Return the discriminant of `x`.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^29 - 17*x - 1, 'alpha')        #...
    # needs sage.libs.pari
sage: K = S.number_field()                             #...
    # needs sage.libs.pari sage.rings.number_field
sage: discriminant(K)                                  #...
    # needs sage.libs.pari sage.rings.number_field
-15975100446626038280218213241591829458737190477345113376757479850566957249523
```

```python
>>> from sage.all import *                           *
```

```python
>>> R = PolynomialRing(QQ, names=('x',')); (x,) = R._first_ngens(1)
>>> S = R.quotient(x**Integer(29) - Integer(17)*x - Integer(1), 'alpha')  ...  
    # needs sage.libs.pari
>>> K = S.number_field()                             #...
    # needs sage.libs.pari sage.rings.number_field
>>> discriminant(K)                                  #...
    # needs sage.libs.pari sage.rings.number_field
-15975100446626038280218213241591829458737190477345113376757479850566957249523
```

`sage.misc.functional.discriminant(x)`

Return the discriminant of `x`.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^29 - 17*x - 1, 'alpha')        #...
    # needs sage.libs.pari
sage: K = S.number_field()                             #...
    # needs sage.libs.pari sage.rings.number_field
sage: discriminant(K)                                  #...
    # needs sage.libs.pari sage.rings.number_field
-15975100446626038280218213241591829458737190477345113376757479850566957249523
```

```python
>>> from sage.all import *                           *
```

```python
>>> R = PolynomialRing(QQ, names=('x',')); (x,) = R._first_ngens(1)
>>> S = R.quotient(x**Integer(29) - Integer(17)*x - Integer(1), 'alpha')  ...  
    # needs sage.libs.pari
>>> K = S.number_field()                             #...
    # needs sage.libs.pari sage.rings.number_field
>>> discriminant(K)                                  #...
    # needs sage.libs.pari sage.rings.number_field
-15975100446626038280218213241591829458737190477345113376757479850566957249523
```

`sage.misc.functional.eta(x)`

Return the value of the \( \eta \) function at `x`, which must be in the upper half plane.
The $\eta$ function is

$$\eta(z) = e^{\pi iz/12} \prod_{n=1}^{\infty} (1 - e^{2\pi inz})$$

EXEMPLARY:

```python
sage: eta(1 + I) # needs sage.symbolic
0.7420487758365647 + 0.1988313702299107*I
```

```python
>>> from sage.all import *
>>> eta(Integer(1) + I) # needs sage.symbolic
0.7420487758365647 + 0.1988313702299107*I
```

sage.misc.functional.fcp(x, var='x')

Return the factorization of the characteristic polynomial of $x$.

EXEMPLARY:

```python
sage: M = MatrixSpace(QQ, 3, 3)  # needs sage.modules
sage: A = M([1,2,3, 4,5,6, 7,8,9])  # needs sage.modules
sage: fcp(A, 'x')  # needs sage.libs.pari sage.modules
x * (x^2 - 15*x - 18)
```

```python
>>> from sage.all import *
>>> M = MatrixSpace(QQ, Integer(3), Integer(3))  # needs sage.modules
>>> A = M([Integer(1),Integer(2),Integer(3), Integer(4),Integer(5),Integer(6),
Integer(7),Integer(8),Integer(9)])
>>> fcp(A, 'x')  # needs sage.libs.pari sage.modules
x * (x^2 - 15*x - 18)
```

sage.misc.functional.gen(x)

Return the generator of $x$.

EXEMPLARY:

```python
sage: R.<x> = QQ[]; R
Univariate Polynomial Ring in x over Rational Field
sage: gen(R)
```

```python
sage: gen(GF(7))
1
```

```python
sage: A = AbelianGroup(1, [23])  # needs sage.groups
sage: gen(A)  # needs sage.groups
```
```python
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1); R
Univariate Polynomial Ring in x over Rational Field
>>> gen(R)
x
>>> gen(GF(Integer(7)))
1
>>> A = AbelianGroup(Integer(1), [Integer(23)])
# needs sage.groups
>>> gen(A)
# needs sage.groups
f
```

### sage.misc.functional.gens(x)

Return the generators of \( x \).

**EXAMPLES:**

```python
sage: R.<x,y> = SR[]
# needs sage.symbolic
sage: R
Multivariate Polynomial Ring in x, y over Symbolic Ring
sage: gens(R)
# needs sage.symbolic
(x, y)
sage: A = AbelianGroup(5, [5,5,7,8,9])
# needs sage.groups
sage: gens(A)
# needs sage.groups
(f0, f1, f2, f3, f4)
```

```python
>>> from sage.all import *
```

```python
>>> R = SR['x, y']; (x, y,) = R._first_ngens(2)
# needs sage.symbolic
>>> R
Multivariate Polynomial Ring in x, y over Symbolic Ring
>>> gens(R)
# needs sage.symbolic
(x, y)
```

```python
>>> A = AbelianGroup(Integer(5), [Integer(5),Integer(5),Integer(7),Integer(8),Integer(9)])
# needs sage.groups
>>> gens(A)
# needs sage.groups
(f0, f1, f2, f3, f4)
```

### sage.misc.functional.hecke_operator(x, n)

Return the \( n \)-th Hecke operator \( T_n \) acting on \( x \).

**EXAMPLES:**

```python
sage: M = ModularSymbols(1,12)
# needs sage.modular
sage: hecke_operator(M,5)
# needs sage.modular
```
Hecke operator $T_5$ on Modular Symbols space of dimension 3 for $\Gamma_0(1)$ of weight 12 with sign 0 over Rational Field

```python
>>> from sage.all import *
>>> M = ModularSymbols(Integer(1), Integer(12))  # needs sage.modular
>>> hecke_operator(M, Integer(5))  # needs sage.modular
Hecke operator $T_5$ on Modular Symbols space of dimension 3 for $\Gamma_0(1)$ of weight 12 with sign 0 over Rational Field
```

```
sage.misc.functional.image(x)

Return the image of $x$.

**EXAMPLES:**
```
sage: M = MatrixSpace(QQ, 3, 3)  # needs sage.modules
sage: A = M([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  # needs sage.modules
sage: image(A)  # needs sage.modules
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)

Return 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.caculus.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
```
```python
>>> from sage.all import *
>>> f = cyclotomic_polynomial(Integer(10))
>>> integral(f)
1/5*x^5 - 1/4*x^4 + 1/3*x^3 - 1/2*x^2 + x
```

```python
sage: integral(sin(x), x)
# needs sage.symbolic
-cos(x)
```

```python
>>> from sage.all import *
>>> integral(sin(x), x)
# needs sage.symbolic
-cos(x)
```

```python
sage: y = var('y')
# needs sage.symbolic
sage: integral(sin(x), y)
# needs sage.symbolic
y*sin(x)
```

```python
>>> from sage.all import *
>>> y = var('y')
# needs sage.symbolic
>>> integral(sin(x), y)
# needs sage.symbolic
y*sin(x)
```

```python
sage: integral(sin(x), x, 0, pi/2)
# needs sage.symbolic
1
sage: sin(x).integral(x, 0, pi/2)
# needs sage.symbolic
1
sage: integral(exp(-x), (x, 1, oo))
# needs sage.symbolic
e^(-1)
```

```python
>>> from sage.all import *
>>> integral(sin(x), x, Integer(0), pi/Integer(2))
# needs sage.symbolic
1
>>> sin(x).integral(x, Integer(0), pi/Integer(2))
# needs sage.symbolic
1
>>> integral(exp(-x), (x, Integer(1), oo))
# needs sage.symbolic
e^(-1)
```

Numerical approximation:
```
sage: h = integral(tan(x)/x, (x, 1, pi/3))
# needs sage.symbolic
```

(continues on next page)
integrate(tan(x)/x, x, 1, 1/3*pi)  # needs sage.symbolic
sage: h.n()  
0.07571599101...

Specific algorithm can be used for integration:

sage: integral(sin(x)^2, x, algorithm='maxima')  # needs sage.symbolic
1/2*x - 1/4*sin(2*x)
sage: integral(sin(x)^2, x, algorithm='sympy')  # needs sage.symbolic
-1/2*cos(x)*sin(x) + 1/2*x

sage.misc.functional.integral_closure(x)
Return the integral closure of x.

EXAMPLES:

sage: integral_closure(QQ)
Rational Field
sage: K.<a> = QuadraticField(5)  
# needs sage.rings.number_field
sage: O2 = K.order(2 * a); O2  
# needs sage.rings.number_field
Order of conductor 4 generated by 2*a in Number Field in a with defining polynomial x^2 - 5 with a = 2.236067977499790?
sage: integral_closure(O2)  
# needs sage.rings.number_field
Maximal Order generated by 1/2*a + 1/2 in Number Field in a with defining polynomial x^2 - 5 with a = 2.236067977499790?
Utilities, Release 10.4

(continued from previous page)

```python
>>> O2 = K.order(Integer(2) * a); O2
# needs sage.rings.number_field
Order of conductor 4 generated by 2*a in Number Field in a
with defining polynomial x^2 - 5 with a = 2.236067977499790?
```

```python
>>> integral_closure(O2)
# needs sage.rings.number_field
Maximal Order generated by 1/2*a + 1/2 in Number Field in a
with defining polynomial x^2 - 5 with a = 2.236067977499790?
```

```python
sage.misc.functional.integrate(x, *args, **kwds)
Return 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:

```python
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
```

```python
>>> from sage.all import *

sage: integral(sin(x), x)
-cos(x)
```

```python
sage: y = var('y')

sage: integral(sin(x), y)
y*sin(x)
```

```python
>>> from sage.all import *

sage: integral(sin(x), x, 0, pi/2)
1
```

(continues on next page)
Utilities, Release 10.4

(sage): \sin(x).\int(x, 0, \pi/2) \quad \# needs sage.symbolic
1

(sage): \int(\exp(-x), (x, 1, \infty)) \quad \# needs sage.symbolic
\exp(-1)

\begin{verbatim}
>>> from sage.all import *

>>> \int(\tan(x)/x, (x, 1, \pi/3)) \quad \# needs sage.symbolic
\integrate(\tan(x)/x, x, 1, 1/3*\pi)

>>> \int(\tan(x)/x, (x, 1, \pi/3)) \quad \# needs sage.symbolic
\integrate(\tan(x)/x, x, 1, 1/3*\pi)

>>> from sage.all import *

>>> \int(\tan(x)/x, (x, Integer(1), \pi/Integer(3))) \quad \# needs sage.symbolic
\integrate(\tan(x)/x, x, 1, 1/3*\pi)

>>> \int(\tan(x)/x, (x, Integer(1), \pi/Integer(3))) \quad \# needs sage.symbolic
\integrate(\tan(x)/x, x, 1, 1/3*\pi)
\end{verbatim}

Numerical approximation:

\begin{verbatim}
(sage): h = \int(\tan(x)/x, (x, 1, \pi/3)) \quad \# needs sage.symbolic
...

(sage): h \quad \# needs sage.symbolic
\integrate(\tan(x)/x, x, 1, 1/3*\pi)

(sage): h.n() \quad \# needs sage.symbolic
0.07571599101...

>>> from sage.all import *

>>> h = \int(\tan(x)/x, (x, Integer(1), \pi/Integer(3))) \quad \# needs sage.symbolic
...

>>> h \quad \# needs sage.symbolic
\integrate(\tan(x)/x, x, 1, 1/3*\pi)

>>> h.n() \quad \# needs sage.symbolic
0.07571599101...
\end{verbatim}

Specific algorithm can be used for integration:

\begin{verbatim}
(sage): \int(\sin(x)^2, x, \text{algorithm}='\text{maxima}') \quad \# needs sage.symbolic
1/2*x - 1/4*\sin(2*x)

(sage): \int(\sin(x)^2, x, \text{algorithm}='\text{sympy}') \quad \# needs sage.symbolic
-1/2*\cos(x)*\sin(x) + 1/2*x

>>> from sage.all import *

>>> \int(\sin(x)^\text{Integer}(2), x, \text{algorithm}='\text{maxima}') \quad \# needs sage.symbolic
1/2*x - 1/4*\sin(2*x)

>>> \int(\sin(x)^\text{Integer}(2), x, \text{algorithm}='\text{sympy}')
\end{verbatim}

(continues on next page)
sage.misc.functional.interval \((a, b)\)

Integers between \(a\) and \(b\) inclusive (\(a\) and \(b\) integers).

**EXAMPLES:**

```python
sage: I = interval(1, 3)
sage: 2 in I
True
sage: 1 in I
True
sage: 4 in I
False
```

```python
>>> from sage.all import *

>>> I = interval(Integer(1), Integer(3))

>>> Integer(2) in I
True

>>> Integer(1) in I
True

>>> Integer(4) in I
False
```

sage.misc.functional.is_even \((x)\)

Return whether or not an integer \(x\) is even, e.g., divisible by 2.

**EXAMPLES:**

```python
sage: is_even(-1)
False
sage: is_even(4)
True
sage: is_even(-2)
True
```

```python
>>> from sage.all import *

>>> is_even(-Integer(1))
False

>>> is_even(-Integer(1))
False

>>> is_even(Integer(4))
True

>>> is_even(-Integer(2))
True
```

sage.misc.functional.is_odd \((x)\)

Return whether or not \(x\) is odd.

This is by definition the complement of `is_even()`.

**EXAMPLES:**

```python
sage: is_odd(-2)
False
sage: is_odd(-3)
True
```

(continues on next page)
sage: is_odd(0)
False
sage: is_odd(1)
True

```python
>>> from sage.all import *
```

```python
>>> is_odd(-Integer(2))
False
>>> is_odd(-Integer(3))
True
>>> is_odd(Integer(0))
False
>>> is_odd(Integer(1))
True
```

`sage.misc.functional.isqrt(x)`

Return an integer square root, i.e., the floor of a square root.

**EXAMPLES:**

```python
sage: isqrt(10)
3
sage: isqrt(10r)
3
```

```python
>>> from sage.all import *
```

```python
>>> isqrt(Integer(10))
3
```

`sage.misc.functional.kernel(x)`

Return the left kernel of $x$.

**EXAMPLES:**

```python
sage: # needs sage.modules
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:
[]
```

```python
>>> from sage.all import *
```

```python
>>> # needs sage.modules
```

```python
>>> M = MatrixSpace(QQ, Integer(3), Integer(2))
>>> A = M([Integer(1),Integer(2), Integer(3),Integer(4), Integer(5),Integer(6)])
>>> kernel(A)
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[]
```
Here are two corner cases:

```python
sage: # needs sage.modules
sage: M = MatrixSpace(QQ, 0, 3)
sage: A = M([])
```

```python
sage: kernel(A)
Vector space of degree 0 and dimension 0 over Rational Field
Basis matrix:
[]
```

```python
sage: kernel(A.transpose()).basis()
```

```python
[ (1, 0, 0),
  (0, 1, 0),
  (0, 0, 1)
]
```

```python
sage: from sage.all import *
```
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
```

```
>>> from sage.all import *

>>> Mod(Integer(2),Integer(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) # needs sage.libs.pari
sage: lift(xmod - Integer(7)) # needs sage.libs.pari
x - 7
```

sage.misc.functional.log(*args, **kwds)
Return the logarithm of the first argument to the base of the second argument which if missing defaults to \( e \).

It calls the log method of the first argument when computing the logarithm, thus allowing the use of logarithm on any object containing a log method. In other words, log works on more than just real numbers.

**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:**
To change the base of the logarithm, add a second parameter:

```
sage: log(1000, 10)
3
```

The synonym `ln` can only take one argument:

```
sage: ln(10)
2.302585092994046
sage: ln(2.718)
0.999896315728952
sage: ln(2.0)
0.693147180559945
sage: ln(-1)
3.141592653589793j
```

You can use `RDF`, `RealField` or `n` to get a numerical real approximation:

```
sage: log(1024, 2)
10
sage: RDF(log(1024, 2))
10.0
```
\[
\frac{1}{2}\log(10)/\log(2)
\]
\[
sage: RDF(\log(10, 4))
1.6609640474436813
\]
\[
sage: \log(10, 2)
\]
\[
\log(10)/\log(2)
\]
\[
sage: n(\log(10, 2))
3.32192809488736
\]
\[
sage: \log(10, e)
\]
\[
\log(10)
\]
\[
sage: n(\log(10, e))
2.30258509299405
\]

```python
>>> from sage.all import *
```

```python
>>> log(Integer(1024), Integer(2))
10
```

```python
>>> RDF(log(Integer(1024), Integer(2)))
10.0
```

```python
>>> # needs sage.symbolic
```

```python
>>> log(Integer(10), Integer(4))
\frac{1}{2}\log(10)/\log(2)
```

```python
>>> RDF(log(Integer(10), Integer(4)))
1.6609640474436813
```

```python
>>> log(Integer(10), Integer(2))
\log(10)/\log(2)
```

```python
>>> n(log(Integer(10), Integer(2)))
3.32192809488736
```

```python
>>> log(Integer(10), e)
\log(10)
```

```python
>>> n(log(Integer(10), e))
2.30258509299405
```

The \texttt{log} function works for negative numbers, complex numbers, and symbolic numbers too, picking the branch with angle between \( -\pi \) and \( \pi \):

```python
sage: \log(-1+0*I) # needs sage.symbolic
I*pi
```

```python
sage: \log(CC(-1)) # needs sage.rings.real_mpfr
3.14159265358979*I
```

```python
sage: \log(-1.0) # needs sage.symbolic
3.14159265358979*I
```

```python
>>> from sage.all import *
```

```python
>>> \log(-Integer(1)+Integer(0)*I) # needs sage.symbolic
I*pi
```

```python
>>> \log(CC(-Integer(1))) # needs sage.rings.real_mpfr
3.14159265358979*I
```

```python
>>> \log(-RealNumber('1.0')) # needs sage.symbolic
3.14159265358979*I
```

Small integer powers are factored out immediately:

1.2. Functional notation
The `hold` parameter can be used to prevent automatic evaluation:

```sage
sage: # needs sage.symbolic
sage: log(-1, hold=True)
log(-1)
sage: log(-1)
I*pi
sage: I.log(hold=True)
log(I)
sage: I.log(hold=True).simplify()
1/2*I*pi
```

For input zero, the following behavior occurs:

```sage
sage: log(0)  #...
-Infinity
sage: log(CC(0))  #...
-\infty
sage: log(0.0)  #...
-\infty
```

```sage
>>> from sage.all import *

>>> # needs sage.symbolic
>>> log(Integer(-1), hold=True)
log(-1)
>>> log(Integer(-1))
I*pi
>>> I.log(hold=True)
log(I)
>>> I.log(hold=True).simplify()
1/2*I*pi
```
The log function also works in finite fields as long as the argument lies in the multiplicative group generated by the base:

```python
sage: # needs sage.libs.pari
sage: F = GF(13); g = F.multiplicative_generator(); g
2
sage: a = F(8)
sage: log(a, g); g^log(a, g)
3
8
sage: log(a, Integer(3))
Traceback (most recent call last):
... 
ValueError: no logarithm of 8 found to base 3 modulo 13
```

The log function also works for p-adics (see documentation for p-adics for more information):

```python
sage: R = Zp(5); R
# needs sage.rings.padics
5-adic Ring with capped relative precision 20
sage: a = R(16); a
# needs sage.rings.padics
1 + 3*5 + O(5^20)
sage: log(a)
# needs sage.rings.padics
3*5 + 3*5^2 + 3*5^4 + 3*5^5 + 3*5^6 + 3*5^7 + 2*5^8 + 5^9 + 5^11 + 2*5^12 + 5^13 + 3*5^15 + 2*5^16 + 4*5^17 + 3*5^18 + 3*5^19 + O(5^20)
```
sage.misc.functional.minimal_polynomial(x, var='x')

Return the minimal polynomial of \( x \).

EXAMPLES:

```
sage: # needs sage.libs.pari sage.modules
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.misc.functional.minpoly(x, var='x')

Return the minimal polynomial of \( x \).

EXAMPLES:

```
sage: # needs sage.libs.pari sage.modules
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.misc.functional.multiplicative_order(x)

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

EXAMPLES:

```python
sage: a = mod(5, 11)
sage: multiplicative_order(a)  # needs sage.libs.pari
5
sage: multiplicative_order(mod(2, 11))  # needs sage.libs.pari
10
sage: multiplicative_order(mod(2, 12))  # needs sage.libs.pari
Traceback (most recent call last):
... ArithmeticError: multiplicative order of 2 not defined since it is not a unit...
```

sage.misc.functional.n(x, prec=None, digits=None, algorithm=None)

Return a numerical approximation of self with prec bits (or decimal digits) of precision.

No guarantee is made about the accuracy of the result.

**Note:** Lower case `n()` is an alias for `numerical_approx()` and may be used as a method.

**INPUT:**

- `prec` - precision in bits
• digits – precision in decimal digits (only used if \texttt{prec} is not given)

• \texttt{algorithm} – which algorithm to use to compute this approximation (the accepted algorithms depend on the object)

If neither \texttt{prec} nor \texttt{digits} is given, the default precision is 53 bits (roughly 16 digits).

EXAMPLES:

```python
sage: # needs \texttt{sage.symbolic}
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: # needs \texttt{sage.rings.real_mpfr}
sage: a = CC(-5).n(prec=40)
sage: b = ComplexField(40)(-5)
sage: a == b
True
sage: parent(a) \texttt{is} parent(b)
True
sage: numerical_approx(9)
9.00000000000000
```

```python
>>> from \texttt{sage.all import *}

>>> # needs \texttt{sage.symbolic}

>>> numerical_approx(pi, \texttt{Integer(10)})
3.1
>>> numerical_approx(pi, digits=\texttt{Integer(10)})
3.141592654
>>> numerical_approx(pi**\texttt{Integer(2)} + e, digits=\texttt{Integer(20)})
12.587886229548403854
>>> n(pi**\texttt{Integer(2)} + e)
12.5878862295484
>>> N(pi**\texttt{Integer(2)} + e)
12.5878862295484
>>> n(pi**\texttt{Integer(2)} + e, digits=\texttt{Integer(50)})
12.587886229548403854194778471228813633070946500941

>>> # needs \texttt{sage.rings.real_mpfr}

>>> a = CC(-\texttt{Integer(5)}).n(prec=\texttt{Integer(40)})

>>> b = ComplexField(\texttt{Integer(40)})(-\texttt{Integer(5)})

>>> a == b
True

>>> parent(a) \texttt{is} parent(b)
True

>>> numerical_approx(\texttt{Integer(9)})
9.00000000000000
```

You can also usually use method notation:
Vectors and matrices may also have their entries approximated:

```
sage: v = vector(RDF, [1, 2, 3])  # needs sage.modules
sage: v.n()  # needs sage.modules
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: # needs sage.modules
sage: v = vector(CDF, [1, 2, 3])
sage: v.n()  # needs sage.modules
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: # needs sage.modules
sage: v = vector(QQ, [1/2, 1/3, 1/4])  # needs sage.modules
sage: n(u, prec=15)  # needs sage.modules
(0.5000, 0.3333, 0.2500)
sage: n(u, digits=5)  # needs sage.modules
(0.50000, 0.33333, 0.25000)
sage: # needs sage.modules
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()  # needs sage.modules
True
sage: u  # needs sage.modules
(0.5000, 0.0000, 0.0000, 0.3333, 0.0000, 0.0000, 0.0000, 0.2500)
sage: # needs sage.modules
sage: A = matrix(QQ, 2, 3, range(6))
sage: A.n()  # needs sage.modules
[0.00000000000000  1.00000000000000  2.00000000000000]
[ 3.00000000000000  4.00000000000000  5.00000000000000]
sage: B = matrix(Integers(12), 3, 8, srange(24))
sage: N(B, digits=2)  # needs sage.modules
[0.00 1.0 2.0 3.0 4.0 5.0 6.0 7.0]
```
>>> from sage.all import *

>>> v = vector(RDF, [Integer(1), Integer(2), Integer(3)])  
# needs sage.modules

>>> v.n()  
# needs sage.modules

(1.00000000000000, 2.00000000000000, 3.00000000000000)

>>> v = vector(CDF, [Integer(1), Integer(2), Integer(3)])

>>> v.n()  

(1.00000000000000, 2.00000000000000, 3.00000000000000)

>>> _.parent()

Vector space of dimension 3 over Complex Field with 53 bits of precision

>>> v.n(prec=Integer(20))

(1.0000, 2.0000, 3.0000)

>>> u = vector(QQ, [Integer(1)/Integer(2), Integer(1)/Integer(3), Integer(1)/Integer(4)],
spars=True)

>>> n(u, prec=Integer(15))  
# needs sage.modules

(0.5000, 0.3333, 0.2500)

>>> n(u, digits=Integer(5))  
# needs sage.modules

(0.50000, 0.33333, 0.25000)

>>> v = vector(QQ, [Integer(1)/Integer(2), Integer(1)/Integer(3), Integer(1)/Integer(4)])

>>> u = v.numerical_approx(digits=Integer(4))

>>> u.is_sparse()

True

>>> u

(0.5000, 0.3333, 0.2500)

>>> A = matrix(QQ, Integer(2), Integer(3), range(Integer(6)))

>>> A.n()

[0.000000000000000 1.00000000000000 2.00000000000000]  
[ 3.00000000000000 4.00000000000000 5.00000000000000]

>>> B = matrix(Integers(Integer(12)), Integer(3), Integer(8), srange(Integer(24)))

>>> N(B, digits=Integer(2))  

[0.00 1.0 2.0 3.0 4.0 5.0 6.0 7.0]  
[ 8.0 9.0 10.1 11.0 1.0 2.0 3.0]  
[ 4.0 5.0 6.0 7.0 8.0 9.0 10.1 11.0]

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)  
# needs sage.symbolic

3.14

sage: y = N(3.14, digits=3)  
# needs sage.symbolic

3.14
Increasing the precision of a floating point number is not allowed:

```python
>>> from sage.all import *
>>> x = N(pi, digits=Integer(3)); x
   # needs sage.symbolic
3.14
>>> y = N(RealNumber('3.14'), digits=Integer(3)); y
   # needs sage.rings.real_mpfr
3.14
>>> x == y
   # needs sage.rings.real_mpfr sage.symbolic
False
>>> x.str(base=Integer(2))
   # needs sage.symbolic
11.001001000100
>>> y.str(base=Integer(2))
   # needs sage.rings.real_mpfr
11.001000111101
```

Increasing the precision of a floating point number is not allowed:

```python
sage: CC(-5).n(prec=100)
   # needs sage.rings.real_mpfr
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)
   # needs sage.rings.real_mpfr
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 70 bits, use at most 53 bits
sage: RealField(24).pi().n()
   # needs sage.rings.real_mpfr
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 53 bits, use at most 24 bits
```

```python
>>> from sage.all import *
>>> CC(-Integer(5)).n(prec=Integer(100))
   # needs sage.rings.real_mpfr
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 100 bits, use at most 53 bits
>>> n(1.3, digits=Integer(20))
   # needs sage.rings.real_mpfr
Traceback (most recent call last):
...
```

(continues on next page)
As an exceptional case, digits=1 usually leads to 2 digits (one significant) in the decimal output (see Issue #11647):

```python
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:

```python
sage: N(pi, prec=2)  #...
3.0
sage: N(3, prec=2)   #...
3.0
sage: N(pi - 3, prec=2)  #...
0.00
```

```python
from sage.all import *
```

```python
>>> N(pi, prec=Integer(2))  # needs sage.symbolic
3.0
>>> N(Integer(3), prec=Integer(2))  # needs sage.rings.real_mpfr
3.0
>>> N(pi - Integer(3), prec=Integer(2))  # needs sage.symbolic
0.00
```
sage.misc.functional.ngens(x)

Return the number of generators of \( x \).

EXAMPLES:

```
sage: R.<x,y> = SR[]; R
# needs sage.symbolic
Multivariate Polynomial Ring in x, y over Symbolic Ring
sage: ngens(R) # ...
2
sage: A = AbelianGroup(5, [5,5,7,8,9]) # ...
# needs sage.groups
sage: ngens(A) # ...
5
sage: ngens(ZZ)
1
```

```
>>> from sage.all import *
>>> R = SR['x, y']; (x, y,) = R._first_ngens(2); R # needs sage.symbolic
Multivariate Polynomial Ring in x, y over Symbolic Ring
>>> ngens(R) # ...
2
>>> A = AbelianGroup(Integer(5), [Integer(5),Integer(5),Integer(7),Integer(8), # needs sage.groups
Integer(9)])
>>> ngens(A) # ...
5
>>> ngens(ZZ)
1
```

sage.misc.functional.norm(x)

Return the norm of \( x \).

For matrices and vectors, this returns the L2-norm. The L2-norm of a vector \( \mathbf{v} = (v_1, v_2, \ldots, v_n) \), 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:

\[
\text{norm}(c) = \text{norm}(a + bi) = c \cdot \overline{c} = a^2 + b^2.
\]

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 \( \mathbb{Z}[i] \) of Gaussian integers, where the norm of each Gaussian integer \( c = a + bi \) is defined as its complex norm.

For vector fields on a pseudo-Riemannian manifold \((M, g)\), the function returns the norm with respect to the metric \( g \):

\[
|v| = \sqrt{g(v, v)}
\]
See also:

- `sage.matrix.matrix2.Matrix.norm()`
- `sage.modules.free_module_element.FreeModuleElement.norm()`
- `sage.rings.complex_double.ComplexDoubleElement.norm()`
- `sage.rings.complex_mpfr.ComplexNumber.norm()`
- `sage.symbolic.expression.Expression.norm()`
- `sage.manifolds.differentiable.vectorfield.VectorField.norm()`

EXAMPLES:

The norm of vectors:

```python
sage: # needs sage.modules sage.symbolic
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", domain='real')
sage: v = vector([a, b, c, d])
sage: norm(v)
sqrt(a^2 + b^2 + c^2 + d^2)
```

```python
>>> from sage.all import *
>>> # needs sage.modules sage.symbolic
>>> z = Integer(1) + Integer(2)*I
>>> norm(vector([z]))
sqrt(5)
>>> v = vector([-Integer(1),Integer(2),Integer(3)])
>>> norm(v)
sqrt(14)
>>> _ = var("a b c d", domain='real')
>>> v = vector([a, b, c, d])
>>> norm(v)
sqrt(a^2 + b^2 + c^2 + d^2)
```

The norm of matrices:

```python
sage: # needs sage.modules sage.symbolic
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
```

```python
>>> from sage.all import *
>>> # needs sage.modules sage.symbolic
```

(continues on next page)
The norm of complex numbers:

```sage
# needs sage.symbolic
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
# needs sage.symbolic
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: v = vector([a, b, c])
sage: bool(norm(v).simplify() == sqrt(a^2 + b^2 + c^2))
True
sage: forget()
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

```python
True
>>> norm(a + b).simplify()
a^2 + 2*a*b + b^2
>>> v = vector([a, b, c])
>>> bool(norm(v).simplify() == sqrt(a**Integer(2) + b**Integer(2) +
    →c**Integer(2)))
True
>>> forget()
```

sage.misc.functional.numerator(x)

Return the numerator of x.

EXAMPLES:

```python
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)

Return a numerical approximation of self with prec bits (or decimal digits) of precision.

No guarantee is made about the accuracy of the result.

**Note:** Lower case \(n()\) is an alias for `numerical_approx()` and may be used as a method.

**INPUT:**

- \(\text{prec}\) – precision in bits
- \(\text{digits}\) – precision in decimal digits (only used if \(\text{prec}\) is not given)
- \(\text{algorithm}\) – which algorithm to use to compute this approximation (the accepted algorithms depend on the object)

If neither \(\text{prec}\) nor \(\text{digits}\) is given, the default precision is 53 bits (roughly 16 digits).

**EXAMPLES:**

```python
sage: # needs sage.symbolic
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: # needs sage.rings.real_mpfr
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
>>> from sage.all import *
>>> numerical_approx(pi, Integer(10))
3.1
>>> numerical_approx(pi, digits=Integer(10))
3.141592654
>>> numerical_approx(pi^Integer(2) + e, digits=Integer(20))
12.587886229548403854
>>> n(pi^Integer(2) + e)
12.5878862295484
>>> N(pi^Integer(2) + e)
12.5878862295484
>>> n(pi^Integer(2) + e, digits=Integer(50))
12.587886229548403854194778471228813633070946500941
>>> # needs sage.rings.real_mpfr
>>> a = CC(-Integer(5)).n(prec=Integer(40))
>>> b = ComplexField(Integer(40))(-Integer(5))
>>> a == b
True
>>> parent(a) is parent(b)
True
>>> numerical_approx(Integer(9))
9.00000000000000
You can also usually use method notation:

sage: (pi^2 + e).n()          # needs sage.symbolic
12.5878862295484
sage: (pi^2 + e).numerical_approx()          # needs sage.symbolic
12.5878862295484

>>> from sage.all import *
>>> (pi**Integer(2) + e).n()          # needs sage.symbolic
12.5878862295484

(continues on next page)
Vectors and matrices may also have their entries approximated:

```
sage: v = vector(RDF, [1, 2, 3])  # needs sage.modules
sage: v.n()  # needs sage.modules
(1.00000000000000, 2.00000000000000, 3.00000000000000)

sage: u = vector(QQ, [1/2, 1/3, 1/4])  # needs sage.modules
sage: n(u, prec=15)  # needs sage.modules
(0.5000, 0.3333, 0.2500)

sage: u = vector(QQ, [1/2, 0, 1/3, 0, 0, 1/4], sparse=True)  # needs sage.modules
sage: u.is_sparse()  # needs sage.modules
True
sage: u
(0.5000, 0.0000, 0.0000, 0.3333, 0.0000, 0.0000, 0.0000, 0.2500)
```

```python
from sage.all import *
```

```
>>> v = vector(RDF, [Integer(1), Integer(2), Integer(3)])  # needs sage.modules
```

(continues on next page)
# needs sage.modules
>>> v = vector(CDF, [Integer(1), Integer(2), Integer(3)])
>>> v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
>>> v.parent()
Vector space of dimension 3 over Complex Field with 53 bits of precision
>>> v.n(prec=Integer(20))
(1.0000, 2.0000, 3.0000)

# needs sage.modules
>>> u = vector(QQ, [Integer(1)/Integer(2), Integer(1)/Integer(3), Integer(1)/Integer(4)])
>>> n(u, prec=Integer(15))
(0.5000, 0.3333, 0.2500)
>>> n(u, digits=Integer(5))
(0.50000, 0.33333, 0.25000)

# needs sage.modules
>>> v = vector(QQ, [Integer(1)/Integer(2), Integer(0), Integer(0), Integer(1)/Integer(3), Integer(0), Integer(0), Integer(0), Integer(1)/Integer(4)], sparse=True)
>>> u = v.numerical_approx(digits=Integer(4))
>>> u.is_sparse()
True
>>> u
(0.5000, 0.0000, 0.0000, 0.3333, 0.0000, 0.0000, 0.0000, 0.2500)

# needs sage.modules
>>> A = matrix(QQ, Integer(2), Integer(3), range(Integer(6)))
>>> A.n()
[0.000000000000000 1.00000000000000 2.00000000000000]
[3.00000000000000 4.00000000000000 5.00000000000000]
>>> B = matrix(Integers(Integer(12)), Integer(3), Integer(8), srange(Integer(24)))
>>> N(B, digits=Integer(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                       # needs sage.symbolic
3.14
sage: y = N(3.14, digits=3); y                      # needs sage.rings.real_mpfr
3.14
sage: x == y                                       # needs sage.rings.real_mpfr sage.symbolic
False
sage: x.str(base=2)                                # needs sage.symbolic
'11.0010010001000'
sage: y.str(base=2)                                # needs sage.rings.real_mpfr
'11.0010010001000'
```
(continues on next page)
Increasing the precision of a floating point number is not allowed:

```python
sage: CC(-5).n(prec=100)  # needs sage.rings.real_mpfr
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)  # needs sage.rings.real_mpfr
Traceback (most recent call last):
  ...TypeError: cannot approximate to a precision of 70 bits, use at most 53 bits

sage: RealField(24).pi().n()  # needs sage.rings.real_mpfr
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 Issue #11647):
In the following example, π and 3 are both approximated to two bits of precision and then subtracted, which kills two bits of precision:

```python
def objgen(x):
    return R, x
```

**EXAMPLES:**

```python
sage: R, x = objgen(FractionField(QQ['x']))
sage: R
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: x
x
```

(continues on next page)
sage.misc.functional.objgens(x)

EXAMPLES:

```python
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)
```

```python
from sage.all import *
```

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

sage.misc.functional.order(x)

Return the order of x.

If x is a ring or module element, this is the additive order of x.

EXAMPLES:

```python
sage: C = CyclicPermutationGroup(10)               # needs sage.groups
sage: order(C)                                     # needs sage.groups
10
sage: F = GF(7)
sage: order(F)
7
```

```python
from sage.all import *
```

```python
sage: C = CyclicPermutationGroup(Integer(10))    # needs sage.groups
sage: order(C)                                     # needs sage.groups
10
sage: F = GF(Integer(7))                           # needs sage.groups
sage: order(F)
7
```

sage.misc.functional.quo(x, y, *args, **kwds)

Return 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:

```python
sage: quotient(5, 6)
5/6
sage: quotient(5., 6.)
```

(continues on next page)
sage.misc.functional.quotient(x, y, *args, **kwds)

Return 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:

```python
sage: quotient(5, 6)
5/6
sage: quotient(5., 6.)
0.8333333333333333
sage: R.<x> = ZZ[]; R
Univariate Polynomial Ring in x over Integer Ring
sage: I = Ideal(R, x^2 + 1)
```

sage.misc.functional.rank(x)

Return the rank of x.

EXAMPLES:

We compute the rank of a matrix:
```python
sage: M = MatrixSpace(QQ, 3, 3)  # needs sage.modules
sage: A = M([1, 2, 3, 4, 5, 6, 7, 8, 9])  # needs sage.modules
sage: rank(A)  # needs sage.modules
2
```

We compute the rank of an elliptic curve:

```python
sage: E = EllipticCurve([0,0,1,-1,0])  # needs sage.schemes
sage: rank(E)  # needs sage.schemes
1
```

```python
sage.misc.functional.regulator(x)

Return the regulator of x.

EXAMPLES:

sage: x = polygen(ZZ, 'x')
sage: regulator(NumberField(x^2 - 2, 'a'))
0.881373587019543
```

```python
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:

```python
sage: # needs sage.symbolic
sage: round(sqrt(2), 2)
1.41
sage: q = round(sqrt(2), 5); q
1.41421
sage: type(q)
<class 'sage.rings.real_double.RealDoubleElement'>
```

```python
sage: q = round(sqrt(2)); q
1
sage: type(q)
<class 'sage.rings.integer.Integer'>
```

```python
sage: round(pi)
3
```

```python
sage: b = 5.4999999999999999
sage: round(b)
5
```

```python
>>> from sage.all import *

>>> # needs sage.symbolic

>>> round(sqrt(Integer(2)), Integer(2))
1.41
```

```python
>>> q = round(sqrt(Integer(2)), Integer(5)); q
1.41421
```

```python
>>> type(q)
<class 'sage.rings.real_double.RealDoubleElement'>
```

```python
>>> q = round(sqrt(Integer(2))); q
1
```

```python
>>> type(q)
<class 'sage.rings.integer.Integer'>
```

```python
>>> round(pi)
3
```

```python
>>> b = RealNumber('5.4999999999999999')
```

```python
>>> round(b)
5
```

This example addresses Issue #23502:

```python
sage: n = round(6); type(n)
<class 'sage.rings.integer.Integer'>
```

```python
>>> from sage.all import *

>>> n = round(Integer(6)); type(n)
<class 'sage.rings.integer.Integer'>
```

Since we use floating-point with a limited range, some roundings can't be performed:

```python
sage: round(sqrt(Integer('1'*1000)), 2)  # needs sage.symbolic
+infinity
```

1.2. Functional notation
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 builtins; builtins.round.

**sage.misc.functional.sqrt** *(x, *args, **kwds)*

**INPUT:**
-  `x` – a number
-  `prec` – integer (default: None): if None, returns an exact square root; otherwise returns a numerical square root if necessary, to the given bits of precision.
-  `extend` – bool (default: True): this is a placeholder, and is always ignored or passed to the sqrt method of `x`, since in the symbolic ring everything has a square root.
-  `all` – bool (default: False): if True, return all square roots of `self`, instead of just one.

**EXAMPLES:**

```
sage: sqrt(4)
2
sage: sqrt(4, all=True)
[2, -2]
sage: # needs sage.symbolic
sage: sqrt(-1)
I
sage: sqrt(2)
sqrt(2)
sage: sqrt(2)^2
2
sage: sqrt(x^2)
sqrt(x^2)
```

```
>>> from sage.all import *
>>> sqrt(Integer(4))
2
>>> sqrt(Integer(4), all=True)
[2, -2]

>>> # needs sage.symbolic
>>> sqrt(-Integer(1))
I
>>> sqrt(Integer(2))
sqrt(2)
>>> sqrt(Integer(2))**Integer(2)
2
>>> sqrt(x**Integer(2))
sqrt(x^2)
```
For a non-symbolic square root root, there are a few options. The best is to numerically approximate afterward:

```
sage: sqrt(2).n() #...
1.41421356237310
```

```
sage: sqrt(2).n(prec=100) #...
1.4142135623730950488016887242
```

```
>>> from sage.all import *
>>> sqrt(Integer(2)).n()  # needs sage.symbolic
1.41421356237310
>>> sqrt(Integer(2)).n(prec=Integer(100))  # needs sage.symbolic
1.4142135623730950488016887242
```

Or one can input a numerical type:

```
sage: sqrt(2.)
1.41421356237310
sage: sqrt(2.000000000000000000000000)
1.41421356237309504880169
sage: sqrt(4.0)
2.00000000000000
```

```
>>> from sage.all import *
>>> sqrt(RealNumber(2.))
1.41421356237310
>>> sqrt(RealNumber(2.000000000000000000000000))
1.41421356237309504880169
>>> sqrt(RealNumber(4.0))
2.00000000000000
```

To prevent automatic evaluation, one can use the `hold` parameter after coercing to the symbolic ring:

```
sage: sqrt(SR(4), hold=True)  #...
sqrt(4)
sage: sqrt(4, hold=True)
Traceback (most recent call last):
  ...
TypeError: ..._do_sqrt() got an unexpected keyword argument 'hold'
```

```
>>> from sage.all import *
>>> sqrt(SR(Integer(4)), hold=True)  # needs sage.symbolic
sqrt(4)
>>> sqrt(Integer(4), hold=True)
Traceback (most recent call last):
  ...
TypeError: ..._do_sqrt() got an unexpected keyword argument 'hold'
```

This illustrates that the bug reported in Issue #6171 has been fixed:

```
sage: a = 1.1
sage: a.sqrt(prec=100)  # this is supposed to fail
```

(continues on next page)
Traceback (most recent call last):
... 
TypeError: ...sqrt() got an unexpected keyword argument 'prec'
sage: sqrt(a, prec=100)  # <@

needs sage.rings.real_mpfr
1.0488088481701515469914535137
sage: sqrt(4.00, prec=250)  # <@

needs sage.rings.real_mpfr
2.0000000000000000000000000000000000000000000000000000000000000000000000000

>>> from sage.all import *
>>> a = RealNumber('1.1')
>>> a.sqrt(prec=Integer(100))  # this is supposed to fail
Traceback (most recent call last):
... 
TypeError: ...sqrt() got an unexpected keyword argument 'prec'

needs sage.rings.real_mpfr
1.0488088481701515469914535137

>>> sqrt(RealNumber('4.00'), prec=Integer(250))  # <@

needs sage.rings.real_mpfr
2.0000000000000000000000000000000000000000000000000000000000000000000000000

One can use numpy input as well:

sage: import numpy  # <@


needs numpy
sage: a = numpy.arange(2,5)  # <@

needs numpy
sage: sqrt(a)  # <@

needs numpy
array([1.41421356, 1.73205081, 2., ... ])

>>> from sage.all import *
>>> import numpy  # <@

needs numpy
>>> a = numpy.arange(Integer(2),Integer(5))  # <@

needs numpy
>>> sqrt(a)  # <@

needs numpy
array([1.41421356, 1.73205081, 2., ... ])

sage.misc.functional.squarefree_part(x)

Return the square free part of \( x \), i.e., a divisor \( z \) such that \( x = z y^2 \), 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
```python
>>> from sage.all import *
>>> squarefree_part(Integer(100))
1
>>> squarefree_part(Integer(12))
3
>>> squarefree_part(Integer(10))
10
>>> squarefree_part(216) # see #8976
6
```
```python
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() #... needs sage.libs.pari
(-9) * (x - 6) * x
```
```python
>>> from sage.all import *
>>> x = QQ['x'].gen(0)
>>> S = squarefree_part(-Integer(9)*x*(x-Integer(6))**Integer(7)*(x-Integer(3))**Integer(2)); S
-9*x^2 + 54*x
>>> S.factor() #... needs sage.libs.pari
(-9) * (x - 6) * x
```
```python
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() #... needs sage.libs.pari
(x - 1) * (x^3 + x + 1)
```
```python
>>> from sage.all import *
>>> f = (x**Integer(3) + x + Integer(1))**Integer(3)*(x-Integer(1)); f
x^10 - x^9 + 3*x^8 + 3*x^5 - 2*x^4 - x^3 - 2*x - 1
>>> g = squarefree_part(f); g
x^4 - x^3 + x^2 - 1
>>> g.factor() #... needs sage.libs.pari
(x - 1) * (x^3 + x + 1)
```
```python
sage.misc.functional.symbolic_prod(expression, *args, **kwds)

Return the symbolic product $\prod_{v=a}^{b}$ 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 product
- b – upper endpoint of the product
- algorithm – (default: 'maxima') one of
  - 'maxima' – use Maxima (the default)
```
- 'giac' – (optional) use Giac
- 'sympy' – use SymPy

• hold – (default: False) if True don’t evaluate

EXAMPLES:

```
sage: # needs sage.symbolic
sage: i, k, n = var('i,k,n')
sage: product(k, k, 1, n)
factorial(n)
sage: product(x + i*(i+1)/2, i, 1, 4)
x^4 + 20*x^3 + 127*x^2 + 288*x + 180
sage: product(i^2, i, 1, 7)
25401600
sage: f = function('f')
sage: product(f(i), i, 1, 7)
f(7)*f(6)*f(5)*f(4)*f(3)*f(2)*f(1)
sage: product(f(i), i, 1, n)
product(f(i), i, 1, n)
sage: assume(k>0)
sage: product(integrate(x^k, x, 0, 1), k, 1, n)
1/factorial(n + 1)
sage: product(f(i), i, 1, n).log().log_expand()
sum(log(f(i)), i, 1, n)
```

```python
>>> from sage.all import *
>>> # needs sage.symbolic
>>> i, k, n = var('i,k,n')
>>> product(k, k, Integer(1), n)
factorial(n)
>>> product(x + i*(i+Integer(1))/Integer(2), i, Integer(1), Integer(4))
x^4 + 20*x^3 + 127*x^2 + 288*x + 180
>>> product(i**Integer(2), i, Integer(1), Integer(7))
25401600
>>> f = function('f')
>>> product(f(i), i, Integer(1), Integer(7))
f(7)*f(6)*f(5)*f(4)*f(3)*f(2)*f(1)
>>> product(f(i), i, Integer(1), n)
product(f(i), i, 1, n)
>>> assume(k=Integer(0))
>>> product(integrate(x**k, x, Integer(0), Integer(1)), k, Integer(1), n)
1/factorial(n + 1)
>>> product(f(i), i, Integer(1), n).log().log_expand()
sum(log(f(i)), i, 1, n)
```

```
sage.misc.functional.symbolic_sum(expression, *args, **kwds)
```

Return 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
- 'sympy' – use SymPy

EXAMPLES:

```python
sage: k, n = var('k,n')
needs sage.symbolic
sage: sum(k, k, 1, n).factor()  # ...
needs sage.symbolic
1/2*(n + 1)*n

>>> from sage.all import *

>>> k, n = var('k,n')  # ...
needs sage.symbolic
>>> sum(k, k, Integer(1), n).factor()  # needs sage.symbolic
1/2*(n + 1)*n

sage: sum(1/k^4, k, 1, oo)  # ...
needs sage.symbolic
1/90*pi^4

>>> from sage.all import *

>>> sum(Integer(1)/k**Integer(4), k, Integer(1), oo)  # needs sage.symbolic
1/90*pi^4

sage: sum(1/k^5, k, 1, oo)  # ...
needs sage.symbolic
zeta(5)

>>> from sage.all import *

>>> sum(Integer(1)/k**Integer(5), k, Integer(1), oo)  # needs sage.symbolic
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:

```python
sage: n = var('n')  # ...
needs sage.symbolic
sage: mylist = [1,2,3,4,5]
sage: sum(mylist[n], n, 0, 3)  # ...
needs sage.symbolic
Traceback (most recent call last):
...
TypeError: unable to convert n to an integer
```
```python
>>> from sage.all import *
>>> n = var('n')  # needs sage.symbolic
>>> mylist = [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]
>>> sum(mylist[n], n, Integer(0), Integer(3))  # needs sage.symbolic
Traceback (most recent call last):
... TypeError: unable to convert n to an integer
Use python sum() instead:
```
```
sage: sum(mylist[n] for n in range(4))
10
```

```python
>>> from sage.all import *
>>> sum(mylist[n] for n in range(Integer(4)))
10
```

Also, only a limited number of functions are recognized in symbolic sums:
```
sage: sum(valuation(n, 2), n, 1, 5)  # needs sage.symbolic
Traceback (most recent call last):
... TypeError: unable to convert n to an integer
```
```
>>> from sage.all import *
>>> sum(valuation(n, Integer(2)), n, Integer(1), Integer(5))  # needs sage.symbolic
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
```
```
>>> from sage.all import *
>>> sum(valuation(n + Integer(1), Integer(2)) for n in range(Integer(5)))
3
```
```
(now back to the Sage sum examples)
```

A well known binomial identity:
```
sage: sum(binomial(n, k), k, 0, n)  # needs sage.symbolic
2^n
```
```
>>> from sage.all import *
>>> sum(binomial(n, k), k, Integer(0), n)  # needs sage.symbolic
2^n
```
```
The binomial theorem:
```
Utilities, Release 10.4

sage: x, y = var('x, y')  # needs sage.symbolic
sage: sum(binomial(n, k) * x^k * y^(n-k), k, 0, n)  # needs sage.symbolic
(x + y)^n

>>> from sage.all import *

sage: sum(k * binomial(n, k), k, 1, n)  # needs sage.symbolic
2^(n - 1)*n

sage: sum((-1)^k * binomial(n, k), k, 0, n)  # needs sage.symbolic
0

sage: sum(2^(-k)/(k*(k+1)), k, 1, oo)  # needs sage.symbolic
-log(2) + 1

Another binomial identity (Issue #7952):

sage: t, k, i = var('t,k,i')  # needs sage.symbolic
sage: sum(binomial(i + t, t), i, 0, k)  # needs sage.symbolic
binomial(k + t + 1, t + 1)

1.2. Functional notation 59
Summing a hypergeometric term:

```sage
sage: sum(binomial(n, k) * factorial(k) / factorial(n+1+k), k, 0, n)  # needs sage.symbolic
1/2*sqrt(pi)/factorial(n + 1/2)
```

```python
>>> from sage.all import *

>>> sum(binomial(n, k) * factorial(k) / factorial(n+Integer(1)+k), k, Integer(0), n)  # needs sage.symbolic
1/2*sqrt(pi)/factorial(n + 1/2)
```

We check a well known identity:

```sage
sage: bool(sum(k^3, k, 1, n) == sum(k, k, 1, n)^2)  # needs sage.symbolic
True
```

```python
True
```

A geometric sum:

```sage
sage: a, q = var('a, q')  # needs sage.symbolic

sage: sum(a*q^k, k, 0, n)  # needs sage.symbolic
(a*q^(n + 1) - a)/(q - 1)
```

```python
>>> a, q = var('a, q')  # needs sage.symbolic

>>> sum(a*q**k, k, Integer(0), n)  # needs sage.symbolic
(a*q^(n + 1) - a)/(q - 1)
```

The geometric series:

```sage
sage: assume(abs(q) < 1)  # needs sage.symbolic

sage: sum(a * q^k, k, 0, oo)  # needs sage.symbolic
-a/(q - 1)
```

```python
>>> assume(abs(q) < Integer(1))  # needs sage.symbolic

>>> sum(a * q**k, k, Integer(0), oo)  # needs sage.symbolic
-a/(q - 1)
```

A divergent geometric series. Don’t forget to forget your assumptions:

```sage
sage: forget()  # needs sage.symbolic
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

```python
sage: assume(q > 1)  # needs sage.symbolic
sage: sum(a * q^k, k, 0, oo)  # needs sage.symbolic
Traceback (most recent call last):
... ValueErr...
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 convertible into a Sage expression.

**sage.misc.functional.transpose(x)**

Return the transpose of x.

**EXAMPLES:**

```python
sage: M = MatrixSpace(QQ, 3, 3)  # ...

\[\begin{bmatrix}
1 & 4 & 7 \\
2 & 5 & 8 \\
3 & 6 & 9 \\
\end{bmatrix}\]
```

```python
>>> from sage.all import *
```

**sage.misc.functional.xinterval(a, b)**

Iterator over the integers between a and b, inclusive.

**EXAMPLES:**

```python
sage: I = xinterval(2,5); I
range(2, 6)

sage: 5 in I
True

sage: 6 in I
False
```

```python
>>> from sage.all import *
```

```python
>>> I = xinterval(Integer(2),Integer(5)); I
range(2, 6)

>>> Integer(5) in I
True

>>> Integer(6) in I
False
```
1.3 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.

1.3.1 Command line oriented API

We’ll start with the simplest usage: setting fixed random number seeds and showing that these lead to reproducible results.

```plaintext
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.

```python
sage: set_random_seed(0)
sage: print(rtest())
(303, -0.266166246380421, 1/2*x^2 - 1/95*x - 1/2, (1,3), [ 1, 0, 0, 1, 1 ], 265625921,
→ 5842, 0.9661911734708414)
sage: set_random_seed(1)
sage: print(rtest())
(978, 0.0557699430711638, -3*x^2 - 1/12, (1,2), [ 0, 0, 1, 1, 0 ], 807447831, 29982,
→ 0.8335077654199736)
sage: set_random_seed(2)
sage: print(rtest())
(207, -0.0141049486533456, 4*x^2 + 1/2, (1,2)(4,5), [ 1, 0, 0, 1, 1 ], 1642898426,
→ 41662, 0.19982565117278328)
sage: set_random_seed(0)
sage: print(rtest())
(303, -0.266166246380421, 1/2*x^2 - 1/95*x - 1/2, (1,3), [ 1, 0, 0, 1, 1 ], 265625921,
→ 5842, 0.9661911734708414)
sage: set_random_seed(1)
sage: print(rtest())
(978, 0.0557699430711638, -3*x^2 - 1/12, (1,2), [ 0, 0, 1, 1, 0 ], 807447831, 29982,
→ 0.8335077654199736)
sage: set_random_seed(2)
sage: print(rtest())
(207, -0.0141049486533456, 4*x^2 + 1/2, (1,2)(4,5), [ 1, 0, 0, 1, 1 ], 1642898426,
→ 41662, 0.19982565117278328)
```

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.)
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.

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.
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: r = ZZ(subsage('random_element(2^200)'))
sage: r
# random
127382861620427462924151488498075119241254209468761367941442
sage: set_random_seed(s)
sage: r == ZZ.random_element(2^200)
True
```

```
>>> from sage.all import *
>>> subsage = Sage()
>>> r = ZZ(subsage('random_element(2^200)'))
>>> r
# random
127382861620427462924151488498075119241254209468761367941442
>>> set_random_seed(s)
>>> 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, 539332, 8.26785106378383, 1.3893337539828183)
sage: set_random_seed(1)
sage: random(), getrandbits(20), uniform(5.0, 10.0), normalvariate(0, 1)
(0.8294022851874259, 624859, 5.77894484361117, -0.4201366826308758)
sage: set_random_seed(0)
sage: random(), getrandbits(20), uniform(5.0, 10.0), normalvariate(0, 1)
(0.111439293741037, 539332, 8.26785106378383, 1.3893337539828183)
```

```
>>> from sage.all import *
>>> subsage = Sage()
>>> r = ZZ(subsage('random_element(2^200)'))
>>> r
# random
127382861620427462924151488498075119241254209468761367941442
>>> set_random_seed(s)
>>> r == ZZ.random_element(2^200)
True
```

```
>>> from sage.all import *
>>> subsage = Sage()
>>> r = ZZ(subsage('random_element(Integer(2)^Integer(200))'))
>>> r
True
```

```
```
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.

### 1.3.2 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(); print(r1)
(303, -0.266166246380421, 1/2*x^2 - 1/95*x - 1/2, (1,3), [ 1, 0, 0, 1, 1 ], 265625921, 5842, 0.9661911734708414)
sage: r2 = rtest(); print(r2)
(105, 0.642309615982449, -x^2 - x - 6, (1,3)(4,5), [ 1, 0, 0, 0, 1 ], 53231108, 77132, 0.001767155077382232)
```
We get slightly different results with an intervening `with seed`.

```python
sage: set_random_seed(0)
sage: r1 == rtest()
True
sage: with seed(1): rtest()
(978, 0.0557699430711638, -3*x^2 - 1/12, (1,2), [ 0, 0, 1, 1, 0 ], 807447831, 29982,˓→ 0.8335077654199736)
sage: r2m = rtest(); r2m
(105, 0.642309615982449, -x^2 - x - 6, (1,3)(4,5), [ 1, 0, 0, 0, 1 ], 53231108, 40267,˓→ 0.001767155077382232)
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:

```python
sage: set_random_seed(0)
sage: r1 == rtest()
True
sage: with seed(1):
    print(rtest())
    print(rtest())
(978, 0.0557699430711638, -3*x^2 - 1/12, (1,2), [ 0, 0, 1, 1, 0 ], 807447831, 29982,˓→ 0.8335077654199736)
(138, -0.0404945051288503, 2*x - 24, (1,2,3), [ 1, 1, 0, 1, 1 ], 1010791326, 91360, 0.˓→ 003332230808068083)
sage: r2m == rtest()
True
```

(continues on next page)
The NTL results after the `with seed` don’t depend on how many NTL random numbers were generated inside the `with seed`.

```python
sage: set_random_seed(0) sage: r1 == rtest() True sage: with seed(1): ....: rtest() (978, 0.0557699430711638, -3*x^2 - 1/12, (1,2), [ 0, 0, 1, 1, 0 ], 807447831, 29982, 0.8335077654199736)
```

(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.)

### 1.3.3 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` method 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:

  ```python
  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

  ```python
  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`).

1.3. Random Number States
• GAP – If you are calling code in GAP that uses random numbers, call `set_seed_gap` at the beginning of your function, like this:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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).

1.3.4 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
```

```
>>> from sage.all import *
>>> from sage.misc.randstate import benchmark_libc, benchmark_mt
>>> timeit('benchmark_libc()')  # random
125 loops, best of 3: 1.95 ms per loop
>>> 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
```

```
>>> from sage.all import *
>>> from sage.misc.randstate import benchmark_libc, benchmark_mt
>>> timeit('benchmark_libc()')  # random
125 loops, best of 3: 1.95 ms per loop
>>> 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:
Utilities, Release 10.4

```python
sage: set_random_seed(0)
sage: current_randstate()
<
sage.misc.randstate.randstate object at 0x...
>
sage: current_randstate().python_random().random()
0.111439293741037

>>> from sage.all import *
>>> set_random_seed(Integer(0))
>>> current_randstate()
<sage.misc.randstate.randstate object at 0x...>
>>> current_randstate().python_random().random()
0.111439293741037
```

`sage.misc.randstate.initial_seed()`

Returns the initial seed used to create the current `randstate`.

**EXAMPLES:**

```python
sage: set_random_seed(42)
sage: initial_seed()
42

>>> from sage.all import *
>>> set_random_seed(Integer(42))
>>> initial_seed()
42
```

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()`).

```python
sage: set_random_seed()
sage: initial_seed()  # random
121030915255244661507561642968348336774

>>> from sage.all import *
>>> set_random_seed()
>>> initial_seed()  # random
121030915255244661507561642968348336774
```

`sage.misc.randstate.random()`

Returns a 31-bit random number. Intended as a drop-in replacement for the libc `random()` function.

**EXAMPLES:**

```python
sage: set_random_seed(31)
sage: from sage.misc.randstate import random
sage: random()
32990711

>>> from sage.all import *
>>> set_random_seed(Integer(31))
>>> from sage.misc.randstate import random
>>> 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:

```python
sage: set_random_seed(1414)
sage: current_randstate().ZZ_seed()
48314508034782595865062786044921182484

>>> from sage.all import *
>>> set_random_seed(Integer(1414))
>>> current_randstate().ZZ_seed()
48314508034782595865062786044921182484
```

c_rand_double()

Returns a random floating-point number between 0 and 1.

EXAMPLES:

```python
sage: set_random_seed(2718281828)
sage: current_randstate().c_rand_double()
0.22437207488974298

>>> from sage.all import *
>>> set_random_seed(Integer(2718281828))
>>> 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:

```python
sage: set_random_seed(1207)
sage: current_randstate().c_random()
2008037228

>>> from sage.all import *
>>> set_random_seed(Integer(1207))
>>> current_randstate().c_random()
2008037228
```

We verify the equivalence mentioned above.

```python
sage: from sage.misc.randstate import random
sage: set_random_seed(1207)
sage: random()
2008037228
```
Utilities, Release 10.4

```python
>>> from sage.all import *
>>> from sage.misc.randstate import random
>>> set_random_seed(Integer(1207))
>>> random()
2008037228
```

**long_seed()**

When called on the current `randstate`, returns a 128-bit Python long suitable for seeding another random number generator.

**EXAMPLES:**

```python
sage: set_random_seed(1618)
sage: current_randstate().long_seed()
2560562797745149508607350947089272595
```

```python
>>> from sage.all import *
>>> set_random_seed(Integer(1618))
>>> current_randstate().long_seed()
2560562797745149508607350947089272595
```

**python_random** *(cls=None, seed=None)*

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`.

**INPUT:**

- `cls` – (optional) a class with the same interface as `random.Random` (e.g. a subclass thereof) to use as the Python RNG interface. Otherwise the standard `random.Random` is used.
- `seed` – (optional) an integer to seed the `random.Random` instance with upon creation; if not specified it is seeded using `ZZ.random_element(1 << 128)`.

**EXAMPLES:**

```python
sage: set_random_seed(5)
sage: rnd = current_randstate().python_random()
sage: rnd.random()
0.013558022446944151
sage: rnd.randrange(1000)
544
```

```python
>>> from sage.all import *
>>> set_random_seed(Integer(5))
>>> rnd = current_randstate().python_random()
>>> rnd.random()
0.013558022446944151
>>> rnd.randrange(Integer(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: set_random_seed(0)
sage: from sage.misc.randstate import randstate
sage: r = randstate(314159)
sage: r.seed()
314159
sage: r.python_random().random()
0.111439293741037
sage: r.seed()
314159

>>> from sage.all import *
>>> set_random_seed(Integer(0))
>>> from sage.misc.randstate import randstate
>>> r = randstate(Integer(314159))
>>> r.seed()
314159
>>> r.python_random().random()
0.111439293741037
>>> r.seed()
314159

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)
14967382633255543447453297768680634540939580077
sage: gap(Integer(35)).SCRRandomString()
[ 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1 ]

>>> from sage.all import *
>>> set_random_seed(Integer(99900000999))
>>> current_randstate().set_seed_gap()
>>> gap.Random(Integer(1), Integer(10)**Integer(50))
14967382633255543447453297768680634540939580077
>>> gap(Integer(35)).SCRRandomString()
[ 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 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

1.3. Random Number States 75
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)
```
```
sage: current_randstate().set_seed_ntl(False)
sage: ntl.ZZ_random(10^40)
1495283511775355459459209288047895196007
```

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:

```python
sage: set_random_seed(5551212)
sage: current_randstate().set_seed_pari()
sage: pari.getrand().type()
't_INT'
```

```python
>>> from sage.all import *
>>> set_random_seed(Integer(5551212))
>>> current_randstate().set_seed_pari()
>>> 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:

```python
sage: set_random_seed(5)
sage: initial_seed()
5
```

```python
>>> from sage.all import *
>>> set_random_seed(Integer(5))
>>> initial_seed()
5
```

1.4 Random Numbers with Python API

AUTHORS:

– Carl Witty (2008-03): new file

This module has the same functions as the Python standard module module{random}, but uses the current sage random number state from module{sage.misc.randstate} (so that it can be controlled by the same global random number seeds).

The functions here are less efficient than the functions in module{random}, because they look up the current random number state on each call.
If you are going to be creating many random numbers in a row, it is better to use the functions in module `sage.misc.randstate` directly.

Here is an example:

(The imports on the next two lines are not necessary, since function `randrange` and function `current_randstate` are both available by default at the code `sage:` prompt; but you would need them to run these examples inside a module.)

```python
sage: from sage.misc.prandom import randrange
sage: from sage.misc.randstate import current_randstate
sage: def test1():
    ....:    return sum([randrange(100) for i in range(100)])
sage: def test2():
    ....:    randrange = current_randstate().python_random().randrange
    ....:    return sum([randrange(100) for i in range(100)])
```

Test2 will be slightly faster than test1, but they give the same answer:

```python
sage: with seed(0): test1()
sage: with seed(0): test2()
sage: with seed(1): test1()
sage: with seed(1): test2()
```

The docstrings for the functions in this file are mostly copied from Python's file `random.py`, so those docstrings are “Copyright (c) 2001, 2002, 2003, 2004, 2005, 2006, 2007 Python Software Foundation; All Rights Reserved” and are available under the terms of the Python Software Foundation License Version 2.
Utilities, Release 10.4

sage.misc.prandom.**betavariate**(alpha, beta)

Beta distribution.

Conditions on the parameters are alpha > 0 and beta > 0. Returned values range between 0 and 1.

**EXAMPLES:**

```python
sage: s = betavariate(0.1, 0.9); s  # random
9.75087916621299e-9
sage: 0.0 <= s <= 1.0
True
sage: s = betavariate(0.9, 0.1); s  # random
0.941890400939253
sage: 0.0 <= s <= 1.0
True
```

```python
>>> from sage.all import *

>>> s = betavariate(RealNumber('0.1'), RealNumber('0.9')); s  # random
9.75087916621299e-9
>>> RealNumber('0.0') <= s <= RealNumber('1.0')
True

>>> s = betavariate(RealNumber('0.9'), RealNumber('0.1')); s  # random
0.941890400939253
>>> RealNumber('0.0') <= s <= RealNumber('1.0')
True
```

sage.misc.prandom.**choice**(seq)

Choose a random element from a non-empty sequence.

**EXAMPLES:**

```python
sage: s = [choice(list(primes(10, 100))) for i in range(5)]; s  # random  #...
[17, 47, 11, 31, 47]

sage: all(t in primes(10, 100) for t in s)  #...
needs sage.libs.pari
True
```

```python
>>> from sage.all import *

>>> s = [choice(list(primes(Integer(10), Integer(100)))) for i in...
[17, 47, 11, 31, 47]

>>> all(t in primes(Integer(10), Integer(100)) for t in s)  #...
# needs sage.libs.pari
True
```

sage.misc.prandom.**expovariate**(lambd)

Exponential distribution.

lambd is 1.0 divided by the desired mean. (The parameter would be called “lambda”, but that is a reserved word in Python.) Returned values range from 0 to positive infinity.

**EXAMPLES:**

```python
sage: sample = [expovariate(0.001) for i in range(3)]; sample  # random
[118.152309288166, 722.261959038118, 45.7190543690470]
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

```python
sage: all(s >= 0.0 for s in sample)
True
sage: sample = [expovariate(1.0) for i in range(3)]; sample # random
[0.404201816061304, 0.735220464997051, 0.201765578600627]
sage: all(s >= 0.0 for s in sample)
True

sage: sample = [expovariate(1000) for i in range(3)]; sample # random
[0.0012068700332283973, 8.340929747302108e-05, 0.00219877067980605]
sage: all(s >= 0.0 for s in sample)
True

>>> from sage.all import *

>>> sample = [expovariate(RealNumber(0.001)) for i in range(Integer(3))];
˓→ sample # random
[118.152309288166, 722.261959038118, 45.7190543690470]
>>> all(s >= RealNumber('0.0') for s in sample)
True

>>> sample = [expovariate(RealNumber('1.0')) for i in range(Integer(3))]; sample
˓→ # random
[0.404201816061304, 0.735220464997051, 0.201765578600627]
>>> all(s >= RealNumber('0.0') for s in sample)
True

>>> sample = [expovariate(Integer(1000)) for i in range(Integer(3))]; sample
˓→ # random
[0.0012068700332283973, 8.340929747302108e-05, 0.00219877067980605]
>>> all(s >= RealNumber('0.0') for s in sample)
True

sage.misc.prandom.gammavariate(alpha, beta)

Gammadistribution. Not the gamma function!

Conditions on the parameters are alpha > 0 and beta > 0.

EXAMPLES:

```python
sage: sample = gammavariate(1.0, 3.0); sample # random
6.58282586130638
sage: sample > 0
True
sage: sample = gammavariate(3.0, 1.0); sample # random
3.07801512341612
sage: sample > 0
True

>>> from sage.all import *

>>> sample = gammavariate(RealNumber('1.0'), RealNumber('3.0')); sample # random
6.58282586130638
>>> sample > Integer(0)
True
>>> sample = gammavariate(RealNumber('3.0'), RealNumber('1.0')); sample # random
3.07801512341612
>>> sample > Integer(0)
True
```
sage.misc.prandom.gauss(\(mu, sigma\))

Gaussian distribution.

\(mu\) is the mean, and \(sigma\) is the standard deviation. This is slightly faster than the normalvariate() function, but is not thread-safe.

EXAMPLES:

```python
sage: [gauss(0, 1) for i in range(3)]  # random
[0.9191011757657915, 0.7744526756246484, 0.8638996866800877]
sage: [gauss(0, 100) for i in range(3)]  # random
[24.916051749154448, -62.99272061579273, -8.1993122536718...]
sage: [gauss(1000, 10) for i in range(3)]  # random
[998.7590700045661, 996.1087338511692, 1010.1256817458031]
```

sage.misc.prandom.getrandbits(\(k\)) \(\rightarrow x\). Generates a long int with \(k\) random bits.

EXAMPLES:

```python
sage: getrandbits(10) in range(2^10)
True
sage: getrandbits(200) in range(2^200)
True
sage: getrandbits(4) in range(2^4)
True
```

sage.misc.prandom.lognormvariate(\(mu, sigma\))

Lognormal distribution.

If you take the natural logarithm of this distribution, you’ll get a normal distribution with mean \(mu\) and standard deviation \(sigma\). \(mu\) can have any value, and \(sigma\) must be greater than zero.

EXAMPLES:

```python
sage: [lognormvariate(100, 10) for i in range(3)]  # random
[2.941035688290246e+37, 2.2257548162070125e+38, 4.142299451717446e+43]
```

---

1.4. Random Numbers with Python API
sage.misc.prandom.normalvariate(mu, sigma)

Normal distribution. mu is the mean, and sigma is the standard deviation.

EXAMPLES:

```
sage: [normalvariate(0, 1) for i in range(3)]  # random
[-1.372558980559407, -1.1701670364898928, 0.04324100555110143]
sage: [normalvariate(0, 100) for i in range(3)]  # random
[37.45695875041769, 159.6347743233298, 124.1029321124009]
sage: [normalvariate(1000, 10) for i in range(3)]  # random
[1008.5303090383741, 989.8624892644895, 985.7728921150242]
```

>>> from sage.all import *
>>> [normalvariate(Integer(0), Integer(1)) for i in range(Integer(3))];
[1.0401699394233033, 1.2722080162636495, 1.0153564009379579, 1.1442323078983077]

sage.misc.prandom.paretovariate(alpha)

Pareto distribution. alpha is the shape parameter.

EXAMPLES:

```
sage: sample = [paretovariate(3) for i in range(1, 5)]; sample # random
[1.0401699394233033, 1.2722080162636495, 1.0153564009379579, 1.1442323078983077]
sage: all(s >= 1.0 for s in sample)
True
```

```python
>>> from sage.all import *
>>> sample = [paretovariate(Integer(3)) for i in range(Integer(1), Integer(5))];
[1.0401699394233033, 1.2722080162636495, 1.0153564009379579, 1.1442323078983077]
>>> all(s >= RealNumber('1.0') for s in sample)
True
```

sage.misc.prandom.randint(a, b)

Return random integer in range [a, b], including both end points.

EXAMPLES:

```
sage: s = [randint(0, 2) for i in range(15)]; s  # random
[0, 1, 0, 0, 1, 0, 2, 0, 2, 1, 2, 0, 2, 2, 0, 2]
sage: all(t in [0, 1, 2] for t in s)
True
sage: -100 <= randint(-100, 10) <= 10
True
```

```python
>>> from sage.all import *
>>> s = [randint(Integer(0), Integer(2)) for i in range(Integer(15))]; s  # random
[0, 1, 0, 0, 1, 0, 2, 0, 2, 1, 2, 0, 2, 0, 2, 2]
>>> all(t in [Integer(0), Integer(1), Integer(2)] for t in s)
True
```
sage.misc.prandom.random()

Get the next random number in the range \([0.0, 1.0)\).

EXAMPLES:

```python
sage: sample = [random() for i in [1 .. 4]]; sample # random
[0.111439293741037, 0.5143475134191677, 0.04468968524815642, 0.332490606442413]
sage: all(0.0 <= s <= 1.0 for s in sample)
True
```

sage.misc.prandom.randrange\((start, stop=None, step=1)\)

Choose a random item from range\((start, stop[, step])\).

This fixes the problem with randint() which includes the endpoint; in Python this is usually not what you want.

EXAMPLES:

```python
sage: s = randrange(0, 100, 11)
sage: 0 <= s < 100
True
sage: s % 11
0
sage: 5000 <= randrange(5000, 5100) < 5100
True
sage: s = [randrange(0, 2) for i in range(15)]
sage: all(t in [0, 1] for t in s)
True
```
sage.misc.prandom.sample(population, k)

Choose k unique random elements from a population sequence.

Return a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).

Members of the population need not be hashable or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.

To choose a sample in a range of integers, use xrange as an argument (in Python 2) or range (in Python 3). This is especially fast and space efficient for sampling from a large population: sample(range(10000000), 60)

EXAMPLES:

>>> s = [randrange(Integer(0), Integer(2)) for i in range(Integer(15))]
>>> all(t in [Integer(0), Integer(1)] for t in s)
True

>>> s = randrange(Integer(0), Integer(1000000), Integer(1000))
>>> Integer(0) <= s < Integer(1000000)
True

>>> s % Integer(1000)
0

>>> Integer(100) <= randrange(-Integer(100), Integer(10)) < Integer(10)
True

from sage.misc.misc import is_sublist
sage: l = ["Here", "I", "come", "to", "save", "the", "day"]
sage: s = sample(l, 3); s # random
['Here', 'to', 'day']
sage: is_sublist(sorted(s), sorted(l))
True

sage: len(s)
3

sage: s = sample(range(2**30), 7); s # random
[357009070, 558990255, 196187132, 752551188, 85926697, 954621491, 624802848]
sage: len(s)
7

sage: all(t in range(2**30) for t in s)
True

(continues on previous page)
sage.misc.prandom.shuffle(x)

x, random=random.random -> shuffle list x in place; return None.

Optional arg random is a 0-argument function returning a random float in [0.0, 1.0); by default, the sage.misc.random.random.

EXAMPLES:

```
sage: shuffle([1 .. 10])
```

sage.misc.prandom.uniform(a, b)

Get a random number in the range [a, b).
Equivalent to code{a + (b-a) * random()}.

EXAMPLES:

```
sage: s = uniform(0, 1); s  # random
0.111439293741037
sage: 0.0 <= s <= 1.0
True
sage: s = uniform(e, pi); s  # random  # needs sage.symbolic
0.5143475134191677*pi + 0.48565248658083227*e
sage: bool(e <= s <= pi)  # needs sage.symbolic
True
```

sage.misc.prandom.vonmisesvariate(mu, kappa)

Circular data distribution.

mu is the mean angle, expressed in radians between 0 and 2*pi, and kappa is the concentration parameter, which must be greater than or equal to zero. If kappa is equal to zero, this distribution reduces to a uniform random angle over the range 0 to 2*pi.

EXAMPLES:
Utilities, Release 10.4

sage: sample = [vonmisesvariate(1.0r, 3.0r) for i in range(1, 5)]; sample # random
[0.898328639355427, 0.6718030007041281, 2.030877524813393, 1.714325253725145]
sage: all(s >= 0.0 for s in sample)
True

```python
>>> from sage.all import *
>>> sample = [vonmisesvariate(1.0, 3.0) for i in range(Integer(1), Integer(5))];
    # random
[0.898328639355427, 0.6718030007041281, 2.030877524813393, 1.714325253725145]
>>> all(s >= RealNumber('0.0') for s in sample)
True
```

```python
sage.misc.pranom.weibullvariate(alpha, beta)
```

Weibull distribution.

alpha is the scale parameter and beta is the shape parameter.

EXAMPLES:

```
sage: sample = [weibullvariate(1, 3) for i in range(1, 5)]; sample # random
[0.49069775546342537, 0.8972185564611213, 0.357573846531942, 0.739377255516847]
sage: all(s >= 0.0 for s in sample)
True
```

```python
>>> from sage.all import *
>>> sample = [weibullvariate(Integer(1), Integer(3)) for i in range(Integer(1), Integer(5))]; sample # random
[0.49069775546342537, 0.8972185564611213, 0.357573846531942, 0.739377255516847]
>>> all(s >= RealNumber('0.0') for s in sample)
True
```

1.5 The Unknown truth value

The Unknown object is used in Sage in several places as return value in addition to True and False, in order to signal uncertainty about or inability to compute the result. Unknown can be identified using `is`, or by catching `UnknownError` from a boolean operation.

**Warning:** Calling `bool()` with Unknown as argument will throw an `UnknownError`. This also means that in the following cases, and, not, and or fail or return a somewhat wrong value:

```
sage: not Unknown # should return Unknown
Traceback (most recent call last):
  ... UnknownError: Unknown does not evaluate in boolean context
sage: Unknown and False # should return False
Traceback (most recent call last):
  ... UnknownError: Unknown does not evaluate in boolean context
sage: Unknown or False # should return Unknown
Traceback (most recent call last):
  ... UnknownError: Unknown does not evaluate in boolean context
```
EXAMPLES:

```python
def func(n):
    if n > 0:
        return True
    elif n < 0:
        return False
    else:
        return Unknown
```

Using direct identification:

```python
for n in [-3, 0, 12]:
    res = func(n)
    if res:
        print("n={} is positive".format(n))
    elif res is False:
        print("n={} is negative".format(n))
    else:
        print("n={} is neither positive nor negative".format(n))
```

1.5. The Unknown truth value
Using `UnknownError`:

```python
>>> from sage.all import *
>>> for n in [-Integer(3), Integer(0), Integer(12)]:
... try:
... if func(n):
...     print("n={} is positive".format(n))
... else:
...     print("n={} is negative".format(n))
... except UnknownError:
...     print("n={} is neither positive nor negative".format(n))
```

AUTHORS:


```python
class sage.misc.unknown.UnknownClass
    Bases: UniqueRepresentation

    The Unknown truth value

    The `Unknown` object is used in Sage in several places as return value in addition to `True` and `False`, in order to signal uncertainty about or inability to compute the result. `Unknown` can be identified using `is`, or by catching `UnknownError` from a boolean operation.

    Warning: Calling `bool()` with `Unknown` as argument will throw an `UnknownError`. This also means that applying `and`, `not`, and `or` to `Unknown` might fail.
```

```python
exception sage.misc.unknown.UnknownError
    Bases: TypeError

    Raised whenever `Unknown` is used in a boolean operation.

    EXAMPLES:
```
1.5. The Unknown truth value

```python
sage: not Unknown
Traceback (most recent call last):
...
UnknownError: Unknown does not evaluate in boolean context

>>> from sage.all import *
>>> not Unknown
Traceback (most recent call last):
...
UnknownError: Unknown does not evaluate in boolean context
```
2.1 Special Base Classes, Decorators, etc.

2.1.1 Abstract methods

```python
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__'
```

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

`is_optional()`

Returns whether an abstract method is optional or not.

EXAMPLES:
Abstract methods

**INPUT:**
- \(f\) – a function
- \(\text{optional}\) – 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:

```python
sage: class A():
    ....:   @abstract_method
    ....:   def my_method(self):
    ....:       """
    ....:       The method :meth:`my_method` computes my_method
    ....:       EXAMPLES:
    ....:       ...
    ....:       """
    ....:       pass
gFalse
```

```python
>>> from sage.all import *
>>> class A():
...
```
The method :meth:`my_method` computes my_method

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

```python
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:

```python
sage: class A():  
    ....:   @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
```

```python
>>> from sage.all import *  
>>> class A():  
    ....:   @abstract_method(optional = True)  
    ....:   def my_method(self):  
```
The method \texttt{my\_method} computes \texttt{my\_method}:

\begin{verbatim}
>>> EXAMPLES::
... pass

>>> A.my\_method
<optional abstract method my\_method at ...>

>>> x = A()
>>> x.my\_method
NotImplemented
\end{verbatim}

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

\begin{verbatim}
sage: if x.my\_method is not NotImplemented:
      x.my\_method()
      else:
      print("x.my\_method is not available.")
x.my\_method is not available.
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> if x.my\_method is not NotImplemented:
    x.my\_method()
    else:
    print("x.my\_method is not available.")
x.my\_method is not available.
\end{verbatim}

\textbf{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:

\begin{verbatim}
sage: x.my\_method[]
# todo: not implemented
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> x.my\_method[]
# todo: not implemented
\end{verbatim}

Instead one needs to do:

\begin{verbatim}
sage: A._my\_method[]
# todo: not implemented
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> A._my\_method[]
# todo: not implemented
\end{verbatim}

This could probably be fixed in \texttt{sage.misc.sageinspect}.

\textbf{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:

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

```python
>>> from sage.all import *
>>> abstract_method(optional = True)
<function abstract_method.<locals>.<lambda> at ...>
>>> abstract_method(optional = True)(version)
<optional abstract method version at ...>
```

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

Returns the required and optional abstract methods of the class

**EXAMPLES:**

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

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

```python
>>> from sage.all import *
>>> class AbstractClass:
...     @abstract_method
```

(continues on next page)
....  def required1(): pass
....:
>>>  @abstract_method(optional = True)
....  def optional2(): pass
....:
>>>  @abstract_method(optional = True)
....  def optional1(): pass
....:
>>>  @abstract_method
....  def required2(): pass

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

2.1.2 Bindable classes

class sage.misc.bindable_classBindableClass

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):
    ....:    class Inner:
    ....:        def __init__(self, *args):
    ....:            print(args)
    ....:        def f(self, *args):
    ....:            print('{} {}'.format(self, args))
    ....:        @staticmethod
    ....:        def f_static(*args):
    ....:            print(args)

sage: outer = Outer()

>>> from sage.all import *
>>> from sage.misc.nested_class import NestedClassMetaclass
>>> class Outer(metaclass=NestedClassMetaclass):
...    class Inner:
...        def __init__(self, *args):
...            print(args)
...        def f(self, *args):
...            print('{} {}'.format(self, args))
...        @staticmethod
...        def f_static(*args):
...            print(args)

>>> 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

>>> from sage.all import *
>>> 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)

>>> from sage.all import *
>>> x = outer.Inner(Integer(1),Integer(2),Integer(3))
(1, 2, 3)
```

This is similar to what happens with a static method:

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

>>> from sage.all import *
>>> outer.f_static(Integer(1),Integer(2),Integer(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)

>>> from sage.all import *
>>> outer.f(Integer(1),Integer(2),Integer(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 ...>>

>>> from sage.all import *
>>> 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)

>>> from sage.all import *
>>> Outer.f(outer, Integer(1),Integer(2),Integer(3))
<__main__.Outer object at ...> (1, 2, 3)
```

*BindableClass* gives this binding behavior to all its subclasses.
```python
sage: from sage.misc.bindable_class import BindableClass
sage: class Outer(metaclass=NestedClassMetaclass):
....:     class Inner(BindableClass):
....:         " some documentation "
....:         def __init__(self, outer, *args):
....:             print("{} {}").format(outer, args)
```

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

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

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

```python
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:

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

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

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

(continues on next page)
Still, documentation works as usual:

```python
class sage.misc.bindable_class.BoundClass(*args):
    Bases: partial

class sage.misc.bindable_class.Inner2:
    Bases: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:BindableClass

        Some documentation for Outer.Inner

    Inner2
        alias of Inner2
```

### 2.1.3 Decorators

Python decorators for use in Sage.

**AUTHORS:**

- Tim Dumol (5 Dec 2009) – initial version.

```python
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:

```python
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'd': 1, 'd': 2}
(3, 4)  
my_fun
```

```python
>>> from sage.all import *
>>> from sage.misc.decorators import decorator_defaults
>>> @decorator_defaults
... def my_decorator(f,*args,**kwds):
...     print(kwds)
...     print(args)
...     print(f.__name__)

>>> @my_decorator
... def my_fun(a,b):
...     return a,b
()
()  
my_fun

>>> @my_decorator(Integer(3),Integer(4),c=Integer(1),d=Integer(2))
... def my_fun(a,b):
...     return a,b
c'd': 1, 'd': 2}
(3, 4)  
my_fun
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
Utilities, Release 10.4

>>> from sage.all import *
>>> from sage.misc.decorators import decorator_keywords
>>> @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)
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(Integer(1))
1

>>> from sage.all import *
>>> @preprocess
... def foo(x):
...     return x
>>> foo(None)
0
>>> foo(Integer(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: @infix_operator('multiply')
....: def dot(a, b):
....:     '''Dot product.''
....:     return a.dot_product(b)
sage: u = vector([1, 2, 3])

(continues on next page)
Utilities

An infix element-wise addition operator:

```python
sage: # needs sage.modules
sage: @infix_operator('+')
...: def eadd(a, b):
...:     return a.parent(sum(i + j for i, j in zip(a, b)))
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:

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

(# needs sage.symbolic)
cos(x)^2
```

(continues on next page)
.. def thendo(a, b):
...     return b(a)

>>> x |thendo| cos |thendo| (lambda x: x**Integer(2))

\[ \cos(x)^2 \]

operators = {'add': {'left': '__add__', 'right': '__radd__'}, 'multiply': {'left': '__mul__', 'right': '__rmul__'}, 'or': {'left': '__or__', 'right': '__ror__'}

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.

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 github issue 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'}
    >>> from sage.all import *
    >>> from sage.misc.decorators import rename_keyword
    >>> r = rename_keyword(deprecation=Integer(13109), color=rgbcolor)

    To deprecate an old keyword:

    sage: r = rename_keyword(deprecation=13109, color=rgbcolor)
    >>> from sage.all import *
    >>> from sage.misc.decorators import rename_keyword
    >>> r.renames
    {'color': 'rgbcolor'}
    sage: loads(dumps(r)).renames
    {'color': 'rgbcolor'}

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

2.1. Special Base Classes, Decorators, etc.
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:

```python
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) # needs sage.symbolic
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)...

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

Demonstrate that the argument description are retained from the decorated function through the special method
Utilities, Release 10.4

(when left unchanged) (see Issue #9976):

```python
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)
FullArgSpec(args=['x'], varargs=None, varkw=None, defaults=None,
            kwonlyargs=[], kwonlydefaults=None, annotations={})
```

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:

```python
sage: P.<x,y> = QQ[]
sage: I = P*[x,y]
sage: sage_getfile(I.interreduced_basis)  # known bug
'/sage/rings/polynomial/multi_polynomial_ideal.py'
sage: sage_getsourcelines(I.interreduced_basis)  #...
    @handle_AA_and_QQbar,
    @singular_gb_standard_options,
    @libsingular_gb_standard_options,
    def interreduced_basis(self):
        ...
        return self.basis.reduced()
    ], ...)
```

```python
>>> from sage.all import *
>>> 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
>>> @diff_arg_dec
... def g(x):
...     return x
>>> g(Integer(1))
3
>>> g(Integer(1), some_def_arg=Integer(4))
5
>>> from sage.misc.sageinspect import sage_getargspec
>>> sage_getargspec(g)
FullArgSpec(args=['x'], varargs=None, varkw=None, defaults=None,
            kwonlyargs=[], kwonlydefaults=None, annotations={})
```

```python
>> from sage.all import *
>> def diff_arg_dec(f):
...     @sage_wraps(f)
...     def new_f(y, some_def_arg=Integer(2)):
...         return f(y+some_def_arg)
...     return new_f
>> @diff_arg_dec
... def g(x):
...     return x
>>> g(Integer(1))
3
>>> g(Integer(1), some_def_arg=Integer(4))
5
>> from sage.misc.sageinspect import sage_getargspec
>> sage_getargspec(g)
FullArgSpec(args=['x'], varargs=None, varkw=None, defaults=None,
            kwonlyargs=[], kwonlydefaults=None, annotations={})
```

2.1. Special Base Classes, Decorators, etc. 105

(continues on next page)
The \( f \) attribute of the decorated function refers to the original function:

```
sage: foo = object()
sage: @sage_wraps(foo)
....: def func():
....:    pass
sage: wrapped = sage_wraps(foo)(func)
sage: wrapped.f is foo
True
```

Demonstrate that \texttt{sage\_wraps} works for non-function callables (Issue \#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'
```

(continues on next page)
"My little method"

```
>>> t = T()
>>> t.g(Integer(2))
4
>>> t.g.__doc__
'My little method'
```

The bug described in Issue #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 manueally entered on the sage prompt
81
```

```python
>>> from sage.all import *
>>> def square(f):
...     @sage_wraps(f)
...     def new_f(x):
...         return f(x)*f(x)
...     return new_f
>>> f = lambda x:x**Integer(2)
>>> g = square(f)
>>> g(Integer(3))  # this line used to fail for some people if these command were manually entered on the sage prompt
81
```

```python
class sage.misc.decorators.specialize(*args, **kwargs)
    Bases: object

    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!

```python
>>> from sage.all import *
>>> f = specialize(Integer(5))(lambda x, y: x+y)
>>> f(Integer(10))
15
>>> f(Integer(5))
10
>>> @specialize("Bon Voyage")
... def greet(greeting, name):
...     print("{0}, {1}!").format(greeting, name))
>>> greet("Monsieur Jean Valjean")
Bon Voyage, Monsieur Jean Valjean!
>>> greet(name = 'Javert')
Bon Voyage, Javert!
```

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

```n
```python
>>> s = suboptions('arrow', size=Integer(2))
>>> s.name
'arrow_'
>>> s.options
{'size': 2}
```
```

### 2.1.4 Constant functions

class sage.misc.constant_function.ConstantFunction
Bases: SageObject

A class for function objects implementing constant functions.

EXAMPLES:
```python
sage: f = ConstantFunction(3)
sage: f
The constant function (...) -> 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 ...: attribute lookup ... 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)?

## 2.1.5 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: NestedClassMetaclas

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 subcripting 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__, __classcon-
tains__, 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.

Note: If a class is put in this metaclass it automatically becomes a new-style class:

```
>>> from sage.all import *
>>> from sage.misc.classcall_metaclass import ClasscallMetaclas
>>> class Foo (metaclass=ClasscallMetaclas): pass
>>> x = Foo(); x
<_main__.Foo object at 0x...>
>>> issubclass(Foo, object)
True
>>> isinstance(Foo, type)
True
```
>>> issubclass(Foo, object)
True
>>> isinstance(Foo, type)
True

sage.misc.classcall_metaclass.timeCall(T, n, *args)

We illustrate some timing when using the classcall mechanism.

EXAMPLES:

```python
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:

```python
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():
    ....:     def __init__(self, i):
    ....:         self.i = i+i1
```

```python
sage: %timeit timeCall(object, Integer(1000)) # not tested
625 loops, best of 3: 41.4 µs per loop
```

```python
>>> i1 = int(Integer(1)); i3 = int(Integer(3)) # don't use Sage's Integer
```

```python
>>> class PRef():
...     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:

```python
sage: class P(metaclass=ClasscallMetaclass):
    ....:     def __init__(self, i):
    ....:         self.i = i+i1
```

```python
sage: %timeit timeCall(PRef, 1000, i3) # not tested
625 loops, best of 3: 420 µs per loop
```

```python
sage: %timeit timeCall(P, 1000, i3) # not tested
625 loops, best of 3: 458 µs per loop
```

```python
>>> from sage.all import *
```

```python
>>> class P(metaclass=ClasscallMetaclass):
...     def __init__(self, i):
...         self.i = i+i1
```

(continues on next page)
For a Cython class (not cdef since they don’t allows metaclasses), the overhead is a little larger:

```python
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:

```python
sage: class PC(object, metaclass=ClasscallMetaClass):
    ... @staticmethod
    ... def __classcall__(cls, i):
    ...     return i+i1
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 use cases, the overhead should be mostly hidden by the computations inside the classcall:

```python
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:

```python
>>> from sage.all import *
>>> %timeit timeCall(C2C.__classcall__, Integer(1000), C2C, i3)  # not tested
625 loops, best of 3: 33 µs per loop
>>> %timeit timeCall(PC.__classcall__, Integer(1000), PC, i3)  # not tested
625 loops, best of 3: 131 µs per loop
```
Utilities, Release 10.4

```python
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
```

```python
>>> from sage.all import *
>>> %timeit timeCall(C2, Integer(1000), i3)  # not tested
625 loops, best of 3: 330 µs per loop
>>> %timeit timeCall(C3, Integer(1000), i3)  # not tested
625 loops, best of 3: 328 µs per loop
```

```python
sage.misc.classcall_metaclass.typecall(cls, *args, **kwds)
```

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:

```python
sage: from sage.misc.classcall_metaclass import typecall
sage: class Foo(): pass
sage: typecall(Foo)
<__main__.Foo object at 0x...>
sage: typecall(list)[]
sage: typecall(Integer, 2)
2
```

```python
>>> from sage.all import *
>>> from sage.misc.classcall_metaclass import typecall
>>> class Foo(): pass
>>> typecall(Foo)
<__main__.Foo object at 0x...>
>>> typecall(list)[]
>>> typecall(Integer, Integer(2))
2
```

### 2.1.6 Metaclass for inheriting comparison functions

This module defines a metaclass `InheritComparisonMetaclass` to inherit comparison functions in Cython extension types. In Python 2, the special methods `__richcmp__`, `__cmp__` and `__hash__` are only inherited as a whole: defining just 1 or 2 of these will prevent the others from being inherited.

To solve this issue, you can use `InheritComparisonMetaclass` as a Cython “metaclass” (see `sage.cpython.cython_metaclass` for the general mechanism). If you do this for an extension type which defines neither `__richcmp__` nor `__cmp__`, then both these methods are inherited from the base class (the MRO is not used).

In Sage, this is in particular used for `sage.structure.element.Element` to support comparisons using the coercion framework.

None of this is relevant to Python classes, which inherit comparison methods anyway.

AUTHOR:
• Jeroen Demeyer (2015-05-22): initial version, see Issue #18329

class sage.misc.inherit_comparison.InheritComparisonClasscallMetaclass

Bases: ClasscallMetaclas, InheritComparisonMetaclas

Combine ClasscallMetaclas with InheritComparisonMetaclas.

class sage.misc.inherit_comparison.InheritComparisonMetaclas

Bases: type

If the type does not define __richcmp__ nor __cmp__, inherit both these methods from the base class. The difference with plain extension types is that comparison is inherited even if __hash__ is defined.

EXAMPLES:

```python
sage: # needs sage.misc.cython
sage: cython(
...:     'from sage.misc.inherit_comparison cimport InheritComparisonMetaclas
...:     cdef class Base():
...:         def __richcmp__(left, right, int op):
...:             print("Calling Base.__richcmp__")
...:             return left is right
...:     cdef class Derived(Base):
...:         def __hash__(self):
...:             return 1
...:     cdef class DerivedWithRichcmp(Base):
...:         @cython.always_allow_keywords(False)
...:         def __getmetaclass__(self):
...:             from sage.misc.inherit_comparison import InheritComparisonMetaclas
...:             return InheritComparisonMetaclas
...:         def __hash__(self):
...:             return 1
...:     a = Derived()
...:     a == a
True
sage: b = DerivedWithRichcmp()
sage: b == b
Calling Base.__richcmp__
True
```

```python
>>> from sage.all import *
```

```python
>>> # needs sage.misc.cython
>>> cython(
...:     'from sage.misc.inherit_comparison cimport InheritComparisonMetaclas
...:     cdef class Base():
...:         def __richcmp__(left, right, int op):
...:             print("Calling Base.__richcmp__")
...:             return left is right
...:
(continues on next page)
```
..:\n>>> cdef class Derived(Base):
...     def __hash__(self):
...         return Integer(1)
..:\n>>> cdef class DerivedWithRichcmp(Base):
...     @cython.always_allow_keywords(False)
...     def __getmetaclass__(self):
...         from sage.misc.inherit_comparison import InheritComparisonMetaClass
...         return InheritComparisonMetaClass
...     def __hash__(self):
...         return Integer(1)
...     \n...     a = Derived()
>>> a == a
True
>>> b = DerivedWithRichcmp()
>>> b == b
Calling Base.__richcmp__
True

2.1.7 Base Class to Support Method Decorators

AUTHOR:

- Martin Albrecht (2009-05): inspired by a conversation with and code by Mike Hansen

```python
from sage.misc.method_decorator import MethodDecorator

class Foo:
    @MethodDecorator
    def bar(self, x):
        return x**2

J = Foo()
J.bar
```

```python
from sage.all import *

class Foo:
    @MethodDecorator
    def bar(self, x):
        return x**Integer(2)

J = Foo()
J.bar
```

2.1. Special Base Classes, Decorators, etc.

115
2.1.8 Multiplex calls to one object to calls to many objects

AUTHORS:

- Martin Albrecht (2011): initial version

```python
class sage.misc.object_multiplexer.Multiplex(*args):
    Bases: object
    Object for a list of children such that function calls on this new object implies that the same function is called on all children.

class sage.misc.object_multiplexer.MultiplexFunction(multiplexer, name):
    Bases: object
    A simple wrapper object for functions that are called on a list of objects.
```

2.1.9 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`

```python
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`.

class sage.misc.fast_methods.Singleton:
    Bases: 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()

    (continues on next page)
```

(continues on next page)
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:

```python
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: orig == type(c)  # same here, only in doctests
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'>
```

```python
>>> from sage.all import *
>>> class D(C):
...     pass
>>> import __main__  # This is only needed ...
>>> __main__.C = C    # ... in doctests
>>> __main__.D = D    # same here, only in doctests
>>> orig = type(c)
>>> orig == type(c)  # same here, only in doctests
False
>>> 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 sage.misc.fast_methods.WithEqualityById
    Bases: object

    Provide hash and equality test based on identity.
```

Note: This class provides the unique representation behaviour of \texttt{UniqueRepresentation}, together with \texttt{CachedRepresentation}.

EXAMPLES:

Any instance of \texttt{UniqueRepresentation} inherits from \texttt{WithEqualityById}.

```python
sage: class MyParent(Parent):
    ....:    def __init__(self, x):
    ....:        self.x = x
    ....:    def __hash__(self):
    ....:        return hash(self.x)
sage: class MyUniqueParent(UniqueRepresentation, MyParent): pass
sage: issubclass(MyUniqueParent, sage.misc.fast_methods.WithEqualityById)
True
```

Inheriting from \texttt{WithEqualityById} provides unique representation behaviour:

```python
>>> from sage.all import *
>>> class MyParent(Parent):
...    def __init__(self, x):
...        self.x = x
...    def __hash__(self):
...        return hash(self.x)
>>> class MyUniqueParent(UniqueRepresentation, MyParent): pass
>>> issubclass(MyUniqueParent, sage.misc.fast_methods.WithEqualityById)
True
```

The hash inherited from \texttt{MyParent} is replaced by a hash that coincides with \texttt{object}'s hash:

```python
sage: hash(a) == hash(a.x)
False
sage: hash(a) == object.__hash__(a)
True
```
```python
>>> from sage.all import *
>>> hash(a) == hash(a.x)
False
>>> 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.

```python
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
```

```python
>>> from sage.all import *
>>> class MyNonUniqueParent(MyUniqueParent):
...     def __eq__(self, other):
...         return self.x**Integer(2) == other.x**Integer(2)
>>> a = MyNonUniqueParent(Integer(1))
>>> d = MyNonUniqueParent(-Integer(1))
>>> a is MyNonUniqueParent(Integer(1))
True
>>> a == d
True
>>> a is d
False
```

### 2.1.10 Attribute and method calling

```python
class sage.misc.call.AttrCallObject(name, args, kwds)
Bases: object
```

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

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

```python
>>> from sage.all import *
>>> f = attrcall('core', Integer(3)); f
*.core(3)
>>> [f(p) for p in Partitions(Integer(5))]  # needs sage.combinat
[[2], [1, 1], [1, 1], [3, 1, 1], [2], [2], [1, 1]]
```

```python
sage.misc.call.call_method(obj, name, *args, **kwds)
    Call the method name on obj.
    This has to exist somewhere in Python!!!

    See also:
    operator.methodcaller() attrcall()

    EXAMPLES:

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

```python
>>> from sage.all import *
>>> from sage.misc.call import call_method
>>> call_method(Integer(1), '__add__', Integer(2))
3
```

### 2.2 Lists and Iteration, etc.

#### 2.2.1 Callable dictionaries

```python
class sage.misc.callable_dict.CallableDict
    Bases: dict
    Callable dictionary.
    This is a trivial subclass of dict with an alternative view as a function.
    Typical use cases involve passing a dictionary \( d \) down to some tool that takes a function as input. The usual idiom in such use cases is to pass the \( d._\text{get}\item \) bound method. A pitfall is that this object is not picklable. When this feature is desired, a CallableDict can be used instead. Note however that, with the current implementation, CallableDict is slightly slower than \( d._\text{get}\item \) (see Issue #6484 for benchmarks, and Issue #18330 for potential for improvement).

    EXAMPLES:

sage: from sage.misc.callable_dict import CallableDict
sage: d = CallableDict({'one': 1, 'zwei': 2, 'trois': 3})
sage: d['zwei']
(continues on next page)
```
In case the input is not in the dictionary, a `ValueError` is raised, for consistency with the function call syntax:

```python
d[1]
```

```
Traceback (most recent call last):
  ...
KeyError: 1
```

```python
d(1)
```

```
Traceback (most recent call last):
  ...
ValueError: 1 is not in dict
```

### 2.2.2 Converting Dictionary

At the moment, the only class contained in this module is a key converting dictionary, which applies some function (e.g. type conversion function) to all arguments used as keys.

**AUTHORS:**


**EXAMPLES:**

A `KeyConvertingDict` will apply a conversion function to all method arguments which are keys:

```python
d = KeyConvertingDict(int)
d["3"] = 42
```

```
[(3, 42)]
```

```python
>>> from sage.all import *
>>> from sage.misc.converting_dict import KeyConvertingDict
>>> d = KeyConvertingDict(int)
```
```python
>>> d["3"] = Integer(42)
>>> list(d.items())
[(3, 42)]
```

This is used e.g. in the result of a variety, to allow access to the result no matter how a generator is identified:

```python
sage: # needs sage.libs.singular sage.rings.number_field
sage: K.<x, y> = QQ[]
sage: I = ideal([x^2 + 2*y - 5, x + y + 3])
```

```python
>>> from sage.all import *
```

```python
class sage.misc.converting_dict.KeyConvertingDict(key_conversion_function, data=None)
    Bases: dict
    A dictionary which automatically applies a conversion to its keys.
    The most common application is the case where the conversion function is the object representing some category, so that key conversion means a type conversion to adapt keys to that category. This allows different representations for keys which in turn makes accessing the correct element easier.
    INPUT:
    • key_conversion_function – a function which will be applied to all method arguments which represent keys.
    • data – optional dictionary or sequence of key-value pairs to initialize this mapping.

    EXAMPLES:
```
```
pop(key, *args)
Remove and retrieve a given element from the dictionary.

INPUT:

• key – A value identifying the element, will be converted.
• default – The value to return if the element is not mapped, optional.

EXAMPLES:

```python
>>> from sage.misc.converting_dict import KeyConvertingDict
sage: d = KeyConvertingDict(int)
sage: d[3] = 42
sage: d.pop("3")
42
sage: d.pop("3", 33)
33
sage: d.pop("3")
Traceback (most recent call last):
  ... KeyError: ...
```

```python
>>> from sage.all import *
>>> from sage.misc.converting_dict import KeyConvertingDict
>>> d = KeyConvertingDict(int)
>>> d[Integer(3)] = Integer(42)
>>> d.pop("3")
42
>>> d.pop("3", Integer(33))
33
>>> d.pop("3")
Traceback (most recent call last):
  ... KeyError: ...
```

setdefault(key, default=None)
Create a given mapping unless there already exists a mapping for that key.

INPUT:

• key – A value identifying the element, will be converted.
• default – The value to associate with the key.

EXAMPLES:

```python
sage: from sage.misc.converting_dict import KeyConvertingDict
sage: d = KeyConvertingDict(int)
```

```python
sage: d.setdefault("3")
[3, None]
```
from sage.all import *
from sage.misc.converting_dict import KeyConvertingDict

d = KeyConvertingDict(int)
d.setdefault("3")
list(d.items())

[(3, None)]

**update** (*args, **kwds*)

Update the dictionary with key-value pairs from another dictionary, sequence of key-value pairs, or keyword arguments.

**INPUT:**

- **key** – A value identifying the element, will be converted.
- **args** – A single dict or sequence of pairs.
- **kwds** – Named elements require that the conversion function accept strings.

**EXAMPLES:**

```
sage: from sage.misc.converting_dict import KeyConvertingDict
d = KeyConvertingDict(int)
d.update([("3",1), (4,2)])
d[3]
1
d.update({"5": 7, "9": 12})
d[9]
12
d = KeyConvertingDict(QQ['x'])
d.update(x=42)
d
d = KeyConvertingDict(int)
d.update([("3",Integer(1)), (Integer(4),Integer(2))])
d[Integer(3)]
1
d.update({"5": Integer(7), "9": Integer(12)})
d[Integer(9)]
12
d = KeyConvertingDict(QQ['x'])
d.update(x=Integer(42))
d
{x: 42}
```
2.2.3 Flatten nested lists

```
Utilities, Release 10.4

2.2.3 Flatten nested lists

sage.misc.flatten.flatten (in_list, ltypes=(<class 'list'>, <class 'tuple'>), max_level=922372036854775807)

Flatten a nested list.

INPUT:

• in_list – a list or tuple
• ltypes – optional list of particular types to flatten
• max_level – the maximum level to flatten

OUTPUT:

a flat list of the entries of in_list

EXAMPLES:

\[\begin{align*}
\text{sage: } & \text{flatten([[[1], [1]], [1], 2])} \\
& [1, 1, 1, 2] \\
\text{sage: } & \text{flatten([[[1], [1], [1]], (4, 5), [[1], [2]])])} \\
& [1, 2, 3, 4, 5, 1, 2] \\
\text{sage: } & \text{flatten([[[1], [1], [1]], (4, 5), [[1], [2]]], max_level=1)} \\
& [1, 2, 3, 4, 5, [1], [2]] \\
\text{sage: } & \text{flatten([[[3], []]], max_level=0)} \\
& [[[3], []]] \\
\text{sage: } & \text{flatten([[[3], []]], max_level=1)} \\
& [[3], []] \\
\text{sage: } & \text{flatten([[[3], []]], max_level=2)} \\
& [3]
\end{align*}\]

```
We give the vector type and then even the vector gets flattened:

```python
sage: tV = sage.modules.vector_rational_dense.Vector_rational_dense
...
# needs sage.modules
sage: flatten((["Hi", 2, vector(QQ, [1,2,3])], (4,5,6)),
         ltypes=(list, tuple, tV))
["Hi", 2, 1, 2, 3, 4, 5, 6]
```

We flatten a finite field.

```python
sage: flatten(GF(5))
[0, 1, 2, 3, 4]
sage: flatten([GF(5)])
[Finite Field of size 5]
sage: tGF = type(GF(5))
sage: flatten([GF(5)], ltypes=(list, tuple, tGF))
[0, 1, 2, 3, 4]
```

Degenerate cases:

```python
sage: flatten([],[])
[]
sage: flatten([[]])
[]
```

```python
>>> from sage.all import *

... from sage.all import *
... tV = sage.modules.vector_rational_dense.Vector_rational_dense
... # needs sage.modules
... flatten((["Hi", Integer(2), vector(QQ, [Integer(1),Integer(2),Integer(3)])],
...         (Integer(4),Integer(5),Integer(6))),
...         ltypes=(list, tuple, tV))
["Hi", 2, 1, 2, 3, 4, 5, 6]
```
2.2.4 Searching a sorted list

This is like the bisect library module, but also returns whether or not the element is in the list, which saves having to do an extra comparison. Also, the function names make more sense.

\[ \text{sage.misc.search.search}(v, x) \]

Return \((\text{True, i})\) where \(i\) is such that \(v[i] == x\) if there is such an \(i\), or \((\text{False, j})\) otherwise, where \(j\) is the position where \(x\) should be inserted so that \(v\) remains sorted.

**INPUT:**

- \(v\) – a list, which is assumed sorted
- \(x\) – Python object

**OUTPUT:**

bool, int

This is implemented using the built-in bisect module.

**EXAMPLES:**

```
>>> from sage.misc.search import search
>>> search([1,4,6,7,8], 6)
(True, 2)
>>> search([1,4,6,7,8], 5)
(False, 2)
>>> search(['a','c','d','h','z'], 'e')
(False, 3)
```

2.2.5 Multidimensional enumeration

**AUTHORS:**

- Joel B. Mohler (2006-10-12)
- William Stein (2006-07-19)
- Jon Hanke

\[ \text{sage.misc.mrange.cantor_product}(*\text{args}, **\text{kwds}) \]

Return an iterator over the product of the inputs along the diagonals a la Cantor pairing.

**INPUT:**

- a certain number of iterables
- repeat – an optional integer. If it is provided, the input is repeated repeat times.
Other keyword arguments are passed to `sage.combinat.integer_lists.invlex.IntegerListsLex`.

EXAMPLES:

```python
sage: from sage.misc.mrange import cantor_product
sage: list(cantor_product([0, 1], repeat=3))
[(0, 0, 0),
 (1, 0, 0),
 (0, 1, 0),
 (0, 0, 1),
 (1, 1, 0),
 (1, 0, 1),
 (0, 1, 1),
 (1, 1, 1)]
sage: list(cantor_product([0, 1], [0, 1, 2, 3]))
[(0, 0), (1, 0), (0, 1), (1, 1), (0, 2), (1, 2), (0, 3), (1, 3)]
```

Infinite iterators are valid input as well:

```python
sage: from itertools import islice
sage: list(islice(cantor_product(ZZ, QQ), 14))
[(0, 0),
 (1, 0),
 (0, 1),
 (-1, 0),
 (1, 1),
 (0, -1),
 (2, 0),
 (-1, 1),
 (1, -1),
 (0, 1/2),
 (-2, 0),
 (2, 1),
 (-1, -1),
 (1, 1/2)]
```

(continues on next page)
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([]))
[()]
```

```
>>> from sage.all import *
>>> list(cartesian_product_iterator([[Integer(1),Integer(2)], [a,b]]))
[(1, a), (1, b), (2, a), (2, b)]
```

sage.misc.mrange.mrange($sizes$, $typ$=class '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, 2, 2, 3]
```

2.2. Lists and Iteration, etc. 129
Examples that illustrate empty multi-ranges:

```python
sage: mrange([5, 3, -2])
[]
sage: mrange([5, 3, 0])
[]
```

This example is not empty, and should not be. See Issue #6561.

```python
sage: mrange([])
[[[]]]
```

AUTHORS:

- Jon Hanke
- William Stein

```python
sage.misc.mrange.mrange_iter(iter_list, typ=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:

```python
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]

>>> from sage.all import *

>>> mrange_iter([range(Integer(3)),[Integer(0),Integer(2)]]
[[0, 0], [0, 2], [1, 0], [1, 2], [2, 0], [2, 2]]

>>> mrange_iter([['Monty','Flying'],['Python','Circus']], tuple)
[('Monty', 'Python'), ('Monty', 'Circus'), ('Flying', 'Python'), ('Flying', 'Circus')]

>>> mrange_iter([[Integer(2),Integer(3),Integer(5),Integer(7)],[Integer(1),
→Integer(2)]]), sum)
[3, 4, 4, 5, 6, 7, 8, 9]

Examples that illustrate empty multi-ranges:

sage: mrange_iter([range(5),range(3),range(0)])
[]

sage: mrange_iter([range(5), range(3), range(-2)])
[]

>>> from sage.all import *

>>> mrange_iter([range(Integer(5)),range(Integer(3)),range(Integer(0))])
[]

>>> mrange_iter([range(Integer(5)), range(Integer(3)), range(-Integer(2))])
[]

This example is not empty, and should not be. See Issue #6561.

sage: mrange_iter([])
[[[]]]

>>> from sage.all import *

>>> mrange_iter([])
[[[]]]

AUTHORS:

• Joel B. Mohler

class sage.misc.mrange.xmrange(sizes, typ=<class 'list'>)

Bases: object

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.
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]
```

(continues on next page)
Examples that illustrate empty multi-ranges.

```python
sage: list(xmrange([5,3,-2]))
[]
sage: list(xmrange([5,3,0]))
[]

>>> from sage.all import *

>>> list(xmrange([Integer(5),Integer(3),-Integer(2)]))
[]

>>> list(xmrange([Integer(5),Integer(3),Integer(0)]))
[]
```

This example is not empty, and should not be. See Issue #6561.

```python
sage: list(xmrange([[]]))
[[[]]]

>>> from sage.all import *

>>> list(xmrange([[]]))
[[[]]]
```

We use a multi-range iterator to iterate through the Cartesian product of sets.

```python
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')

>>> from sage.all import *

>>> X = ['red', 'apple', Integer(389)]
>>> Y = ['orange', 'horse']

>>> 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

```python
class sage.misc.mrange.xmrange_iter(iter_list, typ=<class 'list'>)

Bases: object```

2.2. Lists and Iteration, etc. 133
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([list(range(3)),list(range(2))], tuple);z
xmrange_iter([[[0, 1, 2], [0, 1]], <... 'tuple'>])
sage: for a in z:
....:   print(a)
(0, 0)
(0, 1)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
```

>> from sage.all import *
>> z = xmrange_iter([list(range(Integer(3))),list(range(Integer(2))]), tuple);z
xmrange_iter([[[0, 1, 2], [0, 1]], <... 'tuple'>])
>> 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)]
```

>> from sage.all import *
>> list(xmrange_iter([range(Integer(3)),range(Integer(2))])))
[[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]
>> list(xmrange_iter([range(Integer(3)),range(Integer(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:
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([range(5), range(3), range(-2)]))
[]
sage: list(xmrange_iter([range(5), range(3), range(0)]))
[]
```

This example is not empty, and should not be. See Issue #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')
```

(continues on next page)
AUTHORS:
  • Joel B. Mohler

**cardinality()**

Return the cardinality of this iterator.

**EXAMPLES:**

```python
sage: C = cartesian_product_iterator([range(3), range(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
```

```
>>> from sage.all import *
>>> C = cartesian_product_iterator([range(Integer(3)), range(Integer(4))])
>>> C.cardinality()
12
>>> C = cartesian_product_iterator([ZZ, QQ])
>>> C.cardinality()
+Infinity
>>> C = cartesian_product_iterator([ZZ, [ ]])
>>> C.cardinality()
0
```

### 2.2.6 multi_replace

**sage.misc.multireplace.multiple_replace(dic, text)**

Replace in 'text' all occurrences of any key in the given dictionary by its corresponding value. Returns the new string.

**EXAMPLES:**

```python
sage: from sage.misc.multireplace import multiple_replace
sage: dic = { 'monkey': 'penguin', 'bananas': 'fish' }
sage: txt = 'This monkey really likes the bananas.'
sage: multiple_replace(dic, txt)
'This penguin really likes the fish.'
```

```python
>>> from sage.all import *
>>> from sage.misc.multireplace import multiple_replace
>>> txt = 'This monkey really likes the bananas.'
```
>>> dic = {'monkey': 'penguin', 'bananas': 'fish'}
>>> multiple_replace(dic, txt)
'This penguin really likes the fish.'

2.2.7 Threaded map function

`sage.misc.map_threaded.map_threaded(function, sequence)`

Apply the function to the elements in the sequence by threading recursively through all sub-sequences in the sequence.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: map_threaded(log, [[1,2], [3,e]])
[[0, log(2)], [log(3), 1]]
sage: map_threaded(log, [(1,2), (3,e)])
[[0, log(2)], [log(3), 1]]
sage: map_threaded(N, [[1,2], [3,e]])
[[1.00000000000000, 2.00000000000000], [3.00000000000000, 2.71828182845905]]
sage: map_threaded((x^2).function(x), [[1,2,3,5], [2,10]])
[[1, 4, 9, 25], [4, 100]]
```

map_threaded also works on any object with an apply_map method, e.g., on matrices:

```
sage: map_threaded(lambda x: x^2, matrix([[1,2], [3,4]]))  # needs sage.modules
[ 1  4]
[ 9 16]
```

```
sage: map_threaded(lambda x: x^Integer(2), matrix([[Integer(1),Integer(2)],
                                                 [Integer(3),Integer(4)]]))  # needs sage.modules
[ 1  4]
[ 9 16]
```

AUTHORS:

- William Stein (2007-12); based on feedback from Peter Doyle.
2.2.8 Ranges and the \([1,2,\ldots,n]\) notation

AUTHORS:

- Jeroen Demeyer (2016-02-22): moved here from misc.py and cleaned up.

\texttt{sage.arith.srange.ellipsis_iter}(step=None, *args)

Same as \texttt{ellipsis_range}, but as an iterator (and may end with an Ellipsis).

See also \texttt{ellipsis_range}.

Use \((1,2,\ldots)\) notation.

EXAMPLES:

\begin{verbatim}
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]
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> A = ellipsis_iter(Integer(1),Integer(2),Ellipsis)
>>> [next(A) for _ in range(Integer(10))]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> next(A)
11
>>> A = ellipsis_iter(Integer(1),Integer(3),Integer(5),Ellipsis)
>>> [next(A) for _ in range(Integer(10))]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> A = ellipsis_iter(Integer(1),Integer(2),Ellipsis,Integer(5),Integer(10),Ellipsis)
>>> [next(A) for _ in range(Integer(10))]
[1, 2, 3, 4, 5, 10, 11, 12, 13, 14]
\end{verbatim}

\texttt{sage.arith.srange.ellipsis_range}(step=None, *args)

Return arithmetic sequence determined by the numeric arguments and ellipsis. Best illustrated by examples.

Use \([1,2,\ldots,n]\) notation.

EXAMPLES:

\begin{verbatim}
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)
[]
\end{verbatim}
```python
>>> from sage.all import *

>>> ellipsis_range(Integer(1),Ellipsis,Integer(11),Integer(100))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 100]

>>> ellipsis_range(Integer(0),Integer(2),Ellipsis,Integer(10),Ellipsis,
                  -Integer(20))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

>>> ellipsis_range(Integer(0),Integer(2),Ellipsis,Integer(11),Ellipsis,
                  -Integer(20))
[0, 2, 4, 6, 8, 10, 11, 13, 15, 17, 19]

>>> ellipsis_range(Integer(0),Integer(2),Ellipsis,Integer(11),Ellipsis,
                  -Integer(20), step=Integer(3))
[0, 2, 5, 8, 11, 14, 17, 20]

>>> ellipsis_range(Integer(10),Ellipsis,Integer(0))
[]
```

`sage.arith.srange.srange(*args,**kws)`

Return a list of numbers `start`, `start+step`, ..., `start+k*step`, where `start+k*step < end` and `start+(k+1)*step >= end`.

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.

**INPUT:**

- `start` – number (default: 0)
- `end` – number
- `step` – number (default: 1)
- `universe` -- parent or type where all the elements should live (default: deduce from inputs). This is only used if ``coerce`` is true.
- `coerce` – convert `start`, `end` and `step` to the same universe (either the universe given in `universe` or the automatically detected universe)
- `include_endpoint` – whether or not to include the endpoint (default: False). This is only relevant if `end` is actually of the form `start + k*step` for some integer `k`.
- `endpoint_tolerance` – used to determine whether or not the endpoint is hit for inexact rings (default 1e-5)

**OUTPUT:** a list

**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”.

**See also:**

`xsrange()` – iterator which is used to implement `srange()`.

**EXAMPLES:**

```python
sage: v = srange(5); v
[0, 1, 2, 3, 4]
sage: type(v[2])
<class 'sage.rings.integer.Integer'>
sage: srange(1, 10)
```
Utilities, Release 10.4

(continued from previous page)

```
[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: # needs sage.rings.real_mpfr
sage: srange(1.0, 5.0, include_endpoint=True)
[1.00000000000000, 2.00000000000000, 3.00000000000000, 4.00000000000000, 5.00000000000000]
```

(continues on next page)
Including the endpoint:

```python
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:

```python
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, 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, 0.9999999999999999, 1.1]
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]
```

```python
from sage.all import *

sage: srange(Integer(0), Integer(10), step=Integer(2), include_endpoint=True)
[0, 2, 4, 6, 8, 10]
sage: srange(Integer(0), Integer(10), step=Integer(3), include_endpoint=True)
[0, 3, 6, 9]
```
More examples:

```python
sage: Q = RationalField()
sage: srange(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: srange(1, 5, 0.5)  # needs sage.rings.real_mpfr
[1.00000000000000, 1.50000000000000, 2.00000000000000, 2.50000000000000, 3.00000000000000, 3.50000000000000, 4.00000000000000, 4.50000000000000]
sage: srange(0, 1, 0.4)  # needs sage.rings.real_mpfr
[0.000000000000000, 0.400000000000000, 0.800000000000000]
```

Negative steps are also allowed:

```python
sage: srange(4, 1, -1)
[4, 3, 2]
sage: srange(4, 1, -1/2)
[4, 7/2, 3, 5/2, 2, 3/2]
```

sage.arith.srange.xsrange(start, end=None, step=1, universe=None, coerce=True, include_endpoint=False, endpoint_tolerance=1e-05)

Return an iterator over numbers start, start+step, ..., start+k*step, where start+k*step < end and start+(k+1)*step >= end.

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.

INPUT:

- start – number (default: 0)
Utilities, Release 10.4

- **end** – number
- **step** – number (default: 1)
- **universe** – parent or type where all the elements should live (default: deduce from inputs)
- **coerce** – convert start, end and step to the same universe (either the universe given in universe or the automatically detected universe)
- **include_endpoint** – whether or not to include the endpoint (default: False). This is only relevant if end is actually of the form start + k*step for some integer k.

\`\`endpoint_tolerance\`\` – used to determine whether or not the endpoint is hit for inexact rings (default: 1e-5)

**OUTPUT:** iterator

Unlike `range()`, start and end can be any type of numbers, and the resulting iterator involves numbers of that type.

**Warning:** You need to be careful when using this function over inexact rings: the elements are computed via repeated addition rather than multiplication, which may produce slightly different results. For example:

```sage
sage: sum([1.1] * 10) == 1.1 * 10
False
```

```python
>>> from sage.all import *
>>> sum([RealNumber('1.1')] * Integer(10)) == RealNumber('1.1') * Integer(10)
False
```

Also, the question of whether the endpoint is hit exactly for a given start + k*step is fuzzy for an inexact ring. If start + k*step = end for some k within endpoint_tolerance of being integral, it is considered an exact hit, thus avoiding spurious values falling just below the endpoint.

**EXAMPLES:**

```sage
sage: xsrange(10)
<...generator object at 0x...>
sage: for i in xsrange(1,5):
    print(i)
1
2
3
4
```

```python
>>> from sage.all import *
>>> xsrange(Integer(10))
<...generator object at 0x...>
>>> for i in xsrange(Integer(1),Integer(5)):
    print(i)
1
2
3
4
```

See `srange()` for more examples.
2.2.9 Elements with labels.

This module implements a simple wrapper class for pairs consisting of an “element” and a “label”. For representation purposes (repr, str, latex), this pair behaves like its label, while the element is “silent”. However, these pairs compare like usual pairs (i.e., both element and label have to be equal for two such pairs to be equal). This is used for visual representations of graphs and posets with vertex labels.

```python
class sage.misc.element_with_label.ElementWithLabel(element, label):
    Bases: object

    Auxiliary class for showing/viewing Poset's with non-injective labelings. For hashing and equality testing the resulting object behaves like a tuple `(element, label)`. For any presentation purposes it appears just as label would.

    EXAMPLES:

    sage: # needs sage.combinat sage.graphs
    sage: P = Poset({1: [2,3]})
    sage: labs = {i: P.rank(i) for i in range(1, 4)}
    sage: print(P.plot(element_labels=labs))
     # needs sage.plot
     Graphics object consisting of 6 graphics primitives

    sage: # needs sage.combinat sage.graphs sage.modules
    sage: from sage.misc.element_with_label import ElementWithLabel
    sage: W = WeylGroup("A1")
    sage: P = W.bruhat_poset(facade=True)
    sage: D = W.domain()
    sage: v = D.rho() - D.fundamental_weight(1)
    sage:nP = P.relabel(lambda w: ElementWithLabel(w, w.action(v)))
    sage: list(nP)
    [(0, 0), (0, 0)]
```

```
>> from sage.all import *
>> # needs sage.combinat sage.graphs
>> P = Poset([i: [2,3]])
>> labs = {i: P.rank(i) for i in range(1, 4)}
>> print(P.plot(element_labels=labs))
     # needs sage.plot
     Graphics object consisting of 6 graphics primitives

>> # needs sage.combinat sage.graphs sage.modules
>> from sage.misc.element_with_label import ElementWithLabel
>> W = WeylGroup("A1")
>> P = W.bruhat_poset(facade=True)
>> D = W.domain()
>> v = D.rho() - D.fundamental_weight(1)
>>nP = P.relabel(lambda w: ElementWithLabel(w, w.action(v)))
>> list(nP)
    [(0, 0), (0, 0)]
```
2.3 File and OS Access

2.3.1 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 Issue #13579.

• Jeroen Demeyer (2013-03-17): add atomic_write, see Issue #14292.

• Sebastian Oehms (2021-08-07): add atomic_dir, see Issue #32344

```python
class sage.misc.temporary_file.atomic_dir(target_directory)
Bases: object

Write to a given directory using a temporary directory and then rename it to the target directory. This is for creating a directory whose contents are determined uniquely by the directory name. If multiple threads or processes attempt to create it in parallel, then it does not matter which thread created it. Despite this assumption the contents of the directories differ in the examples for demonstration purpose.

See also atomic_write.

INPUT:

• target_directory – the name of the directory to be written. If it exists then the previous contents will be kept.

EXAMPLES:

```python
sage: from sage.misc.temporary_file import atomic_dir
sage: target_dir = tmp_dir()

sage: with atomic_dir(target_dir) as d:
    ....:    target_file = os.path.join(d.name, 'test')
    ....:    with open(target_file, 'w') as f:
        ....:        _ = f.write("First")
        ....:        f.flush()
        ....:    with atomic_dir(target_dir) as e:
        ....:        target_file2 = os.path.join(e.name, 'test')
        ....:        with open(target_file2, 'w') as g:
        ....:            _ = g.write("Second")
        ....:            g.flush()
    ....:    with open(target_file, 'r') as f:
        ....:        f.read()

"First"

sage: with atomic_dir(target_dir) as d:
    ....:    target_file = os.path.join(d.name, 'test')
    ....:    with open(target_file, 'w') as f:
        ....:        _ = f.write("Third")

sage: target = os.path.join(target_dir, 'test')

sage: with open(target, 'r') as h:
    ....:    h.read()

"Second"

>>> from sage.all import *
>>> from sage.misc.temporary_file import atomic_dir
>>> target_dir = tmp_dir()

... target_file = os.path.join(d.name, 'test')
```

(continues on next page)
... with open(target_file, 'w') as f:
...     _ = f.write("First")
...     f.flush()
... with atomic_dir(target_dir) as e:
...     target_file2 = os.path.join(e.name, 'test')
...     with open(target_file2, 'w') as g:
...         _ = g.write("Second")
...         g.flush()
... with open(target_file, 'r') as f:
...     f.read()
'First'

>>> with atomic_dir(target_dir) as d:
...     target_file = os.path.join(d.name, 'test')
...     with open(target_file, 'w') as f:
...         _ = f.write("Third")

>>> target = os.path.join(target_dir, 'test')

>>> with open(target, 'r') as h:
...     h.read()
'Second'

class sage.misc.temporary_file.atomic_write (target_filename, append=False, mode=438, binary=None, **kwargs)

Bases: object

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 occurred).

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). (Not to be confused with the file opening mode.)
- binary – (boolean, default: True on Python 2, False on Python 3) the underlying file is opened in binary mode. If False then it is opened in text mode and an encoding with which to write the file may be supplied.
- **kwargs – additional keyword arguments passed to the underlying io.open call.

EXAMPLES:

```python
sage: from sage.misc.temporary_file import atomic_write
sage: target_file = tmp_filename()
sage: with open(target_file, 'w') as f:
...     _ = f.write("Old contents")
sage: with atomic_write(target_file) as f:
...     _ = f.write("New contents")
...     f.flush()
...     with open(target_file, 'r') as f2:
...         f2.read()
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

....:    f2.read()
'Old contents'
sage: with open(target_file, 'r') as f:
....:    f.read()
'New contents'

>>> from sage.all import *
>>> from sage.misc.temporary_file import atomic_write
... target_file = tmp_filename()
>>> with open(target_file, 'w') as f:
...    _ = f.write("Old contents")
>>> with atomic_write(target_file) as f:
...    _ = f.write("New contents")
...    f.flush()
...    with open(target_file, 'r') as f2:
...        f2.read()
'Old contents'
>>> with open(target_file, 'r') as f:
...    f.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()
... with open(target_file, w) as f:
...     _ = f.write("Old contents")
... with atomic_write(target_file) as f:
...     f.close()
... with open(f.name, w) as f2:
...     _ = f2.write("Newer contents")
... with open(target_file, r) as f:
...     f.read()
'Newer contents'

>>> from sage.all import *
>>> from sage.misc.temporary_file import atomic_write
... target_file = tmp_filename()
... with open(target_file, w) as f:
...     _ = f.write("Old contents")
... with atomic_write(target_file) as f:
...     f.close()
... with open(f.name, w) as f2:
...     _ = f2.write("Newer contents")
... with open(target_file, r) as f:
...     f.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):
...

(continues on next page)
Some examples of using the `append` option. Note that the file is never opened in "append" mode, it is possible to overwrite existing data:

```python
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: with open(target_file, 'r') as f:
    ....:    f.read()
'Hello World'
sage: with atomic_write(target_file, append=True) as f:
    ....:    _ = f.seek(0)
    ....:    _ = f.write("HELLO")
sage: with open(target_file, 'r') as f:
    ....:    f.read()
'HELLO World'
```

If the target file is a symbolic link, the link is kept and the target of the link is written to:

```python
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: with open(target_file, 'r') as f:
....:   f.read()
'Newest contents'
```

```
>>> from sage.all import *
>>> link_to_target = os.path.join(tmp_dir(), "templink")
>>> os.symlink(target_file, link_to_target)
>>> with atomic_write(link_to_target) as f:
...   _ = f.write("Newest contents")
>>> with open(target_file, 'r') as f:
...   f.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: '{:#o}'.format(os.stat(target_file).st_mode & 0o777)
'0o644'
sage: _ = os.umask(0o077)
sage: with atomic_write(target_file, mode=0o777) as f:
....:   pass
sage: '{:#o}'.format(os.stat(target_file).st_mode & 0o777)
'0o700'
```

```
>>> from sage.all import *
>>> os.chmod(target_file, Integer(0o600))
>>> _ = os.umask(Integer(0o022))
>>> '{:#o}'.format(os.stat(target_file).st_mode & Integer(0o777))
'0o644'
>>> _ = os.umask(Integer(0o077))
>>> with atomic_write(target_file, mode=Integer(0o777)) as f:
...   pass
>>> '{:#o}'.format(os.stat(target_file).st_mode & Integer(0o777))
'0o700'
```

Test writing twice to the same target file. The outermost `with` “wins”:

```
sage: with open(target_file, 'w') as f:
....:   _ = f.write('>>> ')
sage: with atomic_write(target_file, append=True) as f,
....:      _ = atomic_write(target_file, append=True) as g:
....:   _ = f.write("AAA")
....:   f.close()
....:   _ = g.write("BBB")
....:   g.close()
sage: with open(target_file, 'r') as f:
....:   f.read()
'AAA'
```

```
>>> from sage.all import *
>>> with open(target_file, 'w') as f:
...   _ = f.write('>>> ')
(continues on next page)
```
>>> with atomic_write(target_file, append=True) as f, atomic_write(target_file, append=True) as g:
... _ = f.write("AAA"); f.close()
... _ = g.write("BBB"); g.close()
>>> with open(target_file, 'r') as f:
... _ = f.read()
'AAA'

Supplying an encoding means we’re writing the file in “text mode” (in the same sense as `io.open`) and so we must write unicode strings:

```python
sage: target_file = tmp_filename()
sage: with atomic_write(target_file, binary=False, encoding='utf-8') as f:
... _ = f.write(u'Hélas')
sage: import io
sage: with io.open(target_file, encoding='utf-8') as f:
... _ = print(f.read())
Hélas
```

Supplying an encoding in binary mode (or other arguments that don’t make sense to `io.open` in binary mode) is an error:

```python
sage: writer = atomic_write(target_file, binary=True, encoding='utf-8')
sage: with writer as f:
... _ = f.write(u'Hello')
Traceback (most recent call last):
... ValueError: binary mode doesn't take an encoding argument
sage: os.path.exists(writer.tempname)
False
```

```python
>>> from sage.all import *
>>> writer = atomic_write(target_file, binary=True, encoding='utf-8')
>>> with writer as f:
... _ = f.write(u'Hello')
Traceback (most recent call last):
... ValueError: binary mode doesn't take an encoding argument
>>> os.path.exists(writer.tempname)
False
```

```
 sage.misc.temporary_file.spyx_tmp()

The temporary directory used to store pyx files.
```
We cache the result of this function “by hand” so that the same temporary directory will always be returned. A function is used to delay creating a directory until (if) it is needed. The temporary directory is removed when Sage terminates by way of an atexit hook.

```
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: f = open('file_inside_d', 'w')
```

```
>>> from sage.all import *

>>> d = tmp_dir('dir_testing_', '.extension')

>>> d # random output
'/home/username/.sage/temp/hostname/7961/dir_testing_XgRu4p.extension/'

>>> os.chdir(d)

>>> f = open('file_inside_d', 'w')
```

Temporary directories are unaccessible by other users:

```
sage: os.stat(d).st_mode & 0o077
0
sage: f.close()
```

```
>>> from sage.all import *

>>> os.stat(d).st_mode & Integer(0o077)
0

>>> f.close()
```

```
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:**

```python
sage: fn = tmp_filename('just_for_testing_', '.extension')
sage: fn  # random
'/home/username/.sage/temp/hostname/8044/ju$t_for_testing_tVVHsn.extension'
sage: f = open(fn, 'w')
```

Temporary files are unaccessible by other users:

```python
sage: os.stat(fn).st_mode & 0o77
0
sage: f.close()
```

**2.3.2 get_remote_file**

`sage.misc.remote_file.get_remote_file(filename, verbose=True)`

**INPUT:**
- `filename` — the URL of a file on the web, e.g., "http://modular.math.washington.edu/myfile.txt"
- `verbose` — whether to display download status

**OUTPUT:**
This creates a file in the temp directory and returns the absolute path to that file as a `Path` object.

**EXAMPLES:**

```python
sage: url = 'https://www.sagemath.org/files/loadtest.py'
sage: g = get_remote_file(url, verbose=False)  # optional - internet
sage: with open(g) as f: print(f.read())  # optional - internet
print("hi from the net")
print(2 + 3)
```

```python
>>> from sage.all import *
```
2.3.3 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

```python
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)

EXAMPLES:

```python
sage: import sage.misc.messaging
sage: sage.misc.messaging.pushover("Hi, how are you?", user="XXX") # not tested
```
To set default values populate pushover_defaults:

```python
>>> from sage.all import *
>>> import sage.misc.messaging

sage.misc.messaging.pushover("Hi, how are you?", user="XXX") # 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`.

### 2.3.4 Miscellaneous operating system functions

`sage.misc.sage_ostools.have_program(program, path=None)`

Return `True` if a program executable is found in the path given by path.

**INPUT:**

- `program` – a string, the name of the program to check.
- `path` – string or `None`. Paths to search for `program`, separated by `os.pathsep`. If `None`, use the `PATH` environment variable.

**OUTPUT:** `bool`

**EXAMPLES:**

```python
sage: from sage.misc.sage_ostools import have_program
sage: have_program('ls')
True
sage: have_program('there_is_not_a_program_with_this_name')
False
sage: have_program('sh', '/bin')
True
sage: have_program('sage', '/there_is_not_a_path_with_this_name')
False
sage: have_program('there_is_not_a_program_with_this_name', '/bin')
False
```

(continues on next page)
class sage.misc.sage_ostools.redirection

    Bases: object

    Context to implement redirection of files, analogous to the >file or 1>&2 syntax in POSIX shells.

    Unlike the redirect_stdout and redirect_stderr contexts in the Python 3.4 standard library, this acts on the OS level, not on the Python level. This implies that it only works for true files, not duck-type file objects such as StringIO.

    INPUT:

    • source – the file to be redirected
    • dest – where the source file should be redirected to
    • close – (boolean, default: True) whether to close the destination file upon exiting the context. This is only supported if dest is a Python file.

    The source and dest arguments can be either Python files or file descriptors.

    EXAMPLES:

    sage: from sage.misc.sage_ostools import redirection
    sage: fn = tmp_filename()

    >>> from sage.all import *
    >>> from sage.misc.sage_ostools import redirection
    >>> fn = tmp_filename()

    sage: with redirection(sys.stdout, open(fn, 'w')):
    ....:     print("hello world!")
    sage: with open(fn) as f:
    ....:     _ = sys.stdout.write(f.read())
    hello world!

    >>> from sage.all import *
    >>> with redirection(sys.stdout, open(fn, 'w')):
    ...     print("hello world!")
    >>> with open(fn) as f:
    ...     _ = sys.stdout.write(f.read())
    hello world!

    We can do the same using a file descriptor as source:

    sage: fd = sys.stdout.fileno()
    sage: with redirection(fd, open(fn, 'wb')):
    ....:     _ = os.write(fd, b"hello world!\n")
    sage: with open(fn) as f:
    ....:     _ = sys.stdout.write(f.read())
    hello world!

    >>> from sage.all import *
    >>> fd = sys.stdout.fileno()
    >>> with redirection(fd, open(fn, 'wb')):

(continues on next page)
... _ = os.write(fd, b"hello world!\n")

>>> with open(fn) as f:
... _ = sys.stdout.write(f.read())
hello world!

The converse also works:

```
sage: with open(fn, 'w') as f:
..... _ = f.write("This goes to the file\n")
..... with redirection(f, sys.stdout, close=False):
..... _ = f.write("This goes to stdout\n")
..... _ = f.write("This goes to the file again\n")
This goes to stdout
```

```
sage: with open(fn) as f:
..... _ = sys.stdout.write(f.read())
This goes to the file
This goes to the file again
```

```
>>> from sage.all import *
```
```
>>> with open(fn, 'w') as f:
... _ = f.write("This goes to the file\n")
... with redirection(f, sys.stdout, close=False):
... _ = f.write("This goes to stdout\n")
... _ = f.write("This goes to the file again\n")
This goes to stdout
```

```
>>> with open(fn) as f:
... _ = sys.stdout.write(f.read())
This goes to the file
This goes to the file again
```

The same redirection instance can be reused multiple times, provided that close=False:

```
sage: f = open(fn, 'w+')
sage: r = redirection(sys.stdout, f, close=False)
sage: with r:
..... print("Line 1")
sage: with r:
..... print("Line 2")
sage: with f:
..... _ = f.seek(0)
..... _ = sys.stdout.write(f.read())
Line 1
Line 2
```

```
>>> from sage.all import *
```
```
>>> f = open(fn, 'w+')
>>> r = redirection(sys.stdout, f, close=False)
>>> with r:
... print("Line 1")
>>> with r:
... print("Line 2")
>>> with f:
... _ = f.seek(Integer(0))
... _ = sys.stdout.write(f.read())
Line 1
Line 2
```
The redirection also works for subprocesses:

```python
sage: import subprocess
sage: with redirection(sys.stdout, open(fn, 'w')):
    ...:    _ = subprocess.call(['echo', 'hello world'])
sage: with open(fn) as f:
    ...:    _ = sys.stdout.write(f.read())
hello world
```

```python
>>> from sage.all import *
>>> import subprocess
>>> with redirection(sys.stdout, open(fn, 'w')):
    ...:    _ = subprocess.call(['echo', 'hello world'])
>>> with open(fn) as f:
    ...:    _ = sys.stdout.write(f.read())
hello world
```

dest_fd
dest_file
dup_source_fd
source_fd
source_file

```python
sage.misc.sage_ostools.restore_cwd(chdir=None)
```

Context manager that restores the original working directory upon exiting.

INPUT:

- `chdir` – optionally change directories to the given directory upon entering the context manager

EXAMPLES:

```python
sage: from sage.misc.sage_ostools import restore_cwd
sage: import tempfile
sage: orig_cwd = os.getcwd()
sage: with tempfile.TemporaryDirectory() as d:
    ...:    with restore_cwd(d):
    ...:        print(os.getcwd() == orig_cwd)
False
sage: os.getcwd() == orig_cwd
True
```

```python
>>> from sage.all import *
>>> from sage.misc.sage_ostools import restore_cwd
>>> import tempfile
>>> orig_cwd = os.getcwd()
>>> with tempfile.TemporaryDirectory() as d:
    ...:    with restore_cwd(d):
    ...:        print(os.getcwd() == orig_cwd)
False
>>> os.getcwd() == orig_cwd
True
```
2.4 Database Access

2.4.1 Relational (sqlite) Databases Module

INFO:
This module implements classes (SQLDatabase and SQLQuery (pythonic implementation for the user with little or no knowledge of sqlite)) that wrap the basic functionality of sqlite.

Databases are constructed via a triple indexed dictionary called a skeleton. A skeleton should be constructed to fit the following format:

```
| - skeleton -- a triple-indexed dictionary
| - outer key -- table name
| - inner key -- column name
| - inner inner key -- one of the following:
|   - 'primary_key' -- boolean, whether column has been set as primary key
|   - 'index' -- boolean, whether column has been set as index
|   - 'unique' -- boolean, whether column has been set as unique
|   - 'sql' -- one of 'TEXT', 'BOOLEAN', 'INTEGER', 'REAL', or other user defined type
```

An example skeleton of a database with one table, that table with one column:

```
{table1:{col1:{primary_key:False, index:True, sql:'REAL'}}}
```

SQLDatabases can also be constructed via the add, drop, and commit functions. The vacuum function is also useful for restoring hard disk space after a database has shrunk in size.

A SQLQuery can be constructed by providing a query_dict, which is a dictionary with the following sample format:

```
{table_name: 'tbname', display_cols: ['col1', 'col2', 'col3'],
 expression: [col, operator, value]}
```

Finally a SQLQuery also allows the user to directly input the query string for a database, and also supports the '?' syntax by allowing an argument for a tuple of parameters to query.

For full details, please see the tutorial. sage.graphs.graph_database.py is an example of implementing a database class in Sage using this interface.

AUTHORS:
- R. Andrew Ohana (2011-07-16): refactored and rewrote most of the code; merged the Generic classes into the non-Generic versions; changed the skeleton format to include a boolean indicating whether the column stores unique keys; changed the index names so as to avoid potential ambiguity
- Emily A. Kirkman (2008-09-20): added functionality to generate plots and reformat output in show
- Emily A. Kirkman and Robert L. Miller (2007-06-17): initial version

```python
class sage.databases.sql_db.SQLDatabase (filename=None, read_only=None, skeleton=None)
Bases: SageObject

A SQL Database object corresponding to a database file.
```
INPUT:

- `filename` - a string
- `skeleton` - a triple-indexed dictionary:

```
| - outer key -- table name
| - inner key -- column name
| - inner inner key -- one of the following:
|   - ``primary_key`` -- boolean, whether column has been set as primary key
|   - ``index`` -- boolean, whether column has been set as index
|   - ``unique`` -- boolean, whether column has been set as unique
|   - ``sql`` -- one of `'TEXT'`, `'BOOLEAN'`, `'INTEGER'`, `'REAL'`, or other user defined type
```

TUTORIAL:

The `SQLDatabase` class is for interactively building databases intended for queries. This may sound redundant, but it is important. If you want a database intended for quick lookup of entries in very large tables, much like a hash table (such as a Python dictionary), a `SQLDatabase` may not be what you are looking for. The strength of `SQLDatabases` is in queries, searches through the database with complicated criteria.

For example, we create a new database for storing isomorphism classes of simple graphs:

```
sage: D = SQLDatabase()
```

```
>>> from sage.all import *
>>> D = SQLDatabase()
```

In order to generate representatives for the classes, we will import a function which generates all labeled graphs (noting that this is not the optimal way):

```
sage: from sage.groups.perm_gps.partn_ref.refinement_graphs import all_labeled_graphs
```

```
>>> from sage.all import *
>>> from sage.groups.perm_gps.partn_ref.refinement_graphs import all_labeled_graphs
```

We will need a table in the database in which to store the graphs, and we specify its structure with a Python dictionary, each of whose keys is the name of a column:

```
sage: from collections import OrderedDict
sage: table_skeleton = OrderedDict([
    ....: (graph6, {'sql': 'TEXT', 'index': True, 'primary_key': True}),
    ....: (vertices, {'sql': 'INTEGER'}),
    ....: (edges, {'sql': 'INTEGER'})
    ....: ])
```

```
>>> from sage.all import *
>>> from collections import OrderedDict
>>> table_skeleton = OrderedDict([
    ... (graph6, {'sql': 'TEXT', 'index': True, 'primary_key': True}),
    ... (vertices, {'sql': 'INTEGER'})
    ... ])
```

(continues on next page)
Then we create the table:

```python
sage: D.create_table('simon', table_skeleton)
sage: D.show('simon')
```

```
+---------------------+---+---+
| graph6              | vertices | edges |
+---------------------+---+---+
| g6                  | 0  | 0  |
+---------------------+---+---+
```

Now that we have the table, we will begin to populate the table with rows. First, add the graph on zero vertices:

```python
sage: G = Graph()
sage: D.add_row('simon', (G.graph6_string(), 0, 0))
sage: D.show('simon')
```

```
+---------------------+---+---+
| graph6              | vertices | edges |
+---------------------+---+---+
| g6                  | 0  | 0  |
+---------------------+---+---+
```

Next, add the graph on one vertex:

```python
sage: G.add_vertex()
0
sage: D.add_row('simon', (G.graph6_string(), 1, 0))
sage: D.show('simon')
```

```
+---------------------+---+---+
| graph6              | vertices | edges |
+---------------------+---+---+
| g6                  | 0  | 0  |
+---------------------+---+---+
```

Say we want a database of graphs on four or less vertices:
```python
sage: labels = {}

sage: for i in range(2, 5):
    ...:     labels[i] = []
    ...:     for g in all_labeled_graphs(i):
    ...:         g = g.canonical_label(algorithm='sage')
    ...:         if g not in labels[i]:
    ...:             labels[i].append(g)
    ...:     D.add_row('simon', (g.graph6_string(), g.order(), g.size()))

sage: D.show('simon')
```

```
graph6     vertices   edges
------------------------------------------------------------
?          0          0
@          1          0
A?         2          0
A_          2          1
B?         3          0
BG          3          1
BW          3          2
Bw          3          3
C?         4          0
C@         4          1
CB          4          2
CF          4          3
CJ          4          3
CK          4          2
CL          4          4
CN          4          4
C]          4          4
C~          4          6
```

```python
>>> from sage.all import *

```
We can then query the database – let’s ask for all the graphs on four vertices with three edges. We do so by creating two queries and asking for rows that satisfy them both:

```python
sage: Q = SQLQuery(D, {table_name:'simon', 'display_cols':['graph6'], 'expression':['vertices','=',4]})
sage: Q2 = SQLQuery(D, {table_name:'simon', 'display_cols':['graph6'], 'expression':['edges','=',3]})
sage: Q = Q.intersect(Q2)
sage: len(Q.query_results())
3
sage: Q.query_results()  # random
[('CF', 'CF'), ('CJ', 'CJ'), ('CL', 'CL')]
```

```python
>>> from sage.all import *
>>> Q = SQLQuery(D, {table_name:'simon', 'display_cols':['graph6'], 'expression':['vertices','=',Integer(4)]})
>>> Q2 = SQLQuery(D, {table_name:'simon', 'display_cols':['graph6'], 'expression':['edges','=',Integer(3)]})
>>> Q = Q.intersect(Q2)
>>> len(Q.query_results())
3
>>> Q.query_results()  # random
[('CF', 'CF'), ('CJ', 'CJ'), ('CL', 'CL')]
```

NOTE: The values of `display_cols` are always concatenated in intersections and unions.

Of course, we can save the database to file. Here we use a temporary directory that we clean up afterwards:

```python
sage: import tempfile
sage: d = tempfile.TemporaryDirectory()
sage: dbpath = os.path.join(d.name, 'simon.db')
sage: D.save(dbpath)
```

```python
>>> from sage.all import *
>>> import tempfile
>>> d = tempfile.TemporaryDirectory()
>>> dbpath = os.path.join(d.name, 'simon.db')
>>> D.save(dbpath)
```

Now the database’s hard link is to this file, and not the temporary db file. For example, let’s say we open the same file with another class instance. We can load the file as an immutable database:

```python
sage: E = SQLDatabase(dbpath)
sage: E.show('simon')
```

<table>
<thead>
<tr>
<th>graph6</th>
<th>vertices</th>
<th>edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>@</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>A^-</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

(continues on next page)
>>> from sage.all import *
>>> E = SQLDatabase(dbpath)
>>> E.show('simon')
<table>
<thead>
<tr>
<th>graph6</th>
<th>vertices</th>
<th>edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>@</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A^</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>A_</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>B_</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>BG</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>BW</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Bw</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>C^</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>CJ</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>CK</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>CL</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>CN</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>C]</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>C~</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>C-</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

sage: E.drop_table('simon')
Traceback (most recent call last):
... 
RuntimeError: Cannot drop tables from a read only database.

Call cleanup() on the temporary directory to, well, clean it up:

sage: d.cleanup()

>>> from sage.all import *
>>> d.cleanup()

add_column(table_name, col_name, col_dict, default='NULL')
Add a column named col_name to table table_name, whose data types are described by col_dict. The format for this is:

```python
col1: {'primary_key': False, 'unique': True, 'index': True, 'sql': 'REAL'}
```

**INPUT:**

- col_dict – a dictionary:
  - key – column name
    - inner key – one of the following:
      - primary_key – boolean, whether column has been set as primary key
      - index – boolean, whether column has been set as index
      - unique – boolean, whether column has been set as unique
      - sql – one of 'TEXT', 'BOOLEAN', 'INTEGER', 'REAL', or other user defined type

**EXAMPLES:**

```python
sage: from collections import OrderedDict
sage: MonicPolys = SQLDatabase()

sage: MonicPolys.create_table('simon', OrderedDict([(n, {'sql': 'INTEGER', '__index__': True})]))

sage: for n in range(20): MonicPolys.add_row('simon', (n,))

sage: MonicPolys.add_column('simon', n_squared, {'sql': 'INTEGER', '__index__': False}, 0)

sage: MonicPolys.show('simon')

<table>
<thead>
<tr>
<th>n</th>
<th>n_squared</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```python
>>> from sage.all import *
>>> from collections import OrderedDict

>>> MonicPolys = SQLDatabase()

>>> MonicPolys.create_table('simon', OrderedDict([(n, {'sql': 'INTEGER', '__index__': True})]))
```

(continues on next page)
add_data (table_name, rows, entry_order=None)

INPUT:

- rows — a list of tuples that represent one row of data to add (types should match col types in order)
- entry_order — an ordered list or tuple overrides normal order with user defined order

EXAMPLES:

```python
>>> from sage.all import *
>>>
>>> DB = SQLDatabase()
>>> DB.create_table('simon', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
>>> DB.add_rows('simon', [(0,0),(1,1),(1,2)])
>>> DB.add_rows('simon', [(0,0),(4,0),(5,1)], [b2,a1])
>>> cur = DB.get_cursor()
>>> (cur.execute('select * from simon')).fetchall()
[(0, 0), (1, 1), (1, 2), (0, 0), (0, 4), (1, 5)]
```

add_row (table_name, values, entry_order=None)

```
Add the row described by values to the table table_name. Values should be a tuple, of same length and
```
order as columns in given table.

NOTE:
If values is of length one, be sure to specify that it is a tuple of length one, by using a comma, e.g.:

```
sage: values = (6,)
```

EXAMPLES:

```
sage: from sage.all import *
>>> values = (Integer(6),)
```

```
add_rows (table_name, rows, entry_order=None)
```

INPUT:
- rows – a list of tuples that represent one row of data to add (types should match col types in order)
- entry_order – an ordered list or tuple overrides normal order with user defined order

EXAMPLES:

```
sage: DB = SQLDatabase()
sage: DB.create_table('simon', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
sage: DB.add_row('simon', (0, 1))
sage: cur = DB.get_cursor()
sage: (cur.execute('select * from simon')).fetchall()
[(0, 1)]
```

```
add_rows(('simon',), [(0, 0), (1, 1), (1, 2)])
```

```
commit()
```

Commit changes to file.
EXAMPLES:

```python
def create_table(table_name, table_skeleton):
    """Create a new table in the database."
    To create a table, a column structure must be specified. The form for this is a Python dict, for example:

    {"col1": {'sql':'INTEGER', 'index':False, 'unique':True, 'primary_key':False}, ...
    "create_table('graph6', {\'
    "create_table('vertices', {\'

    INPUT:
    • table_name – a string
    • table_skeleton – a double-indexed dictionary
      – outer key – column name
        • inner key – one of the following:
          • primary_key – boolean, whether column has been set as primary key
          • index – boolean, whether column has been set as index
          • unique – boolean, whether column has been set as unique
          • sql – one of 'TEXT', 'BOOLEAN', 'INTEGER', 'REAL', or other user defined type

    NOTE:
    Some SQL features, such as automatically incrementing primary key, require the full word 'INTEGER', not just 'INT'.

    EXAMPLES:
```
delete_rows (query)

Use a SQLQuery instance to modify (delete rows from) the database.

SQLQuery must have no join statements. (As of now, you can only delete from one table at a time – ideas and patches are welcome).

To remove all data that satisfies a SQLQuery, send the query as an argument to delete_rows. Be careful, test your query first.

Recommended use: have some kind of row identification primary key column that you use as a parameter in the query. (See example below).

**INPUT:**

- query – a SQLQuery (Delete the rows returned when query is run).

**EXAMPLES:**

```python
>>> from sage.all import *
>>> from collections import OrderedDict
>>> D = SQLDatabase()
>>> table_skeleton = OrderedDict([
... ('graph6', {'sql':'TEXT', 'index':True, 'primary_key':True}),
... ('vertices', {'sql':'INTEGER'}),
... ('edges', {'sql':'INTEGER'})
... ])
>>> D.create_table('simon', table_skeleton)
>>> D.show('simon')
graph6 vertices edges
------------------------------------------------------------
```

```python
>>> from collections import
>>> from sage.all import
>>> from collections import OrderidDict
>>> D = SQLDatabase()
>>> table_skeleton = OrderedDict(
... ('id', {'sql':'INTEGER', 'primary_key':True, 'index':True}),
... ('a1', {'sql':'bool'}),
... ('b2', {'sql':'int'})
... )
>>> D.create_table('lucy', table_skeleton)
>>> D.show('lucy')
id a1 b2
------------------------------------------------------------
```

```python
sage: Q = SQLQuery(DB, {'table_name': 'lucy', 'display_cols': ['id', 'a1', 'b2'], 'expression': ['id', '>=', 3]})
sage: DB.delete_rows(Q)
sage: DB.show('lucy')
```
Utilities, Release 10.4

(continued from previous page)

```python
>>> from sage.all import *
>>> from collections import OrderedDict
>>> DB = SQLDatabase()
>>> DB.create_table('lucy', OrderedDict([... ('id', {'sql':'INTEGER', 'primary_key':True, 'index':True}),
... ('a1', {'sql':'bool'}),
... ('b2', {'sql':'int'})]))
>>> DB.add_rows('lucy', [(Integer(0),Integer(1),Integer(1)),(Integer(1),
     →Integer(1),Integer(4)),(Integer(2),Integer(0),Integer(7)),(Integer(3),
     →Integer(1),Integer(384)), (Integer(4),Integer(1),Integer(978932))],['id','a1 ↔','b2'])
>>> DB.show('lucy')

<table>
<thead>
<tr>
<th>id</th>
<th>a1</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>
```

```python
>>> Q = SQLQuery(DB, {table_name:'lucy', display_cols:['id','a1','b2'],
     →expression:[['id','>=',Integer(3)]])
>>> DB.delete_rows(Q)
>>> DB.show('lucy')

<table>
<thead>
<tr>
<th>id</th>
<th>a1</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
```

**drop_column** *(table_name, col_name)*

Drop the column *col_name* from table *table_name*.

**EXAMPLES:**

```python
sage: MonicPolys = SQLDatabase()
sage: MonicPolys.create_table('simon', {'n':{'sql':'INTEGER', 'index':True}})
sage: for n in range(20): MonicPolys.add_row('simon', (n,))
sage: MonicPolys.add_column('simon', 'n_squared', {'sql':'INTEGER'}, 0)
sage: MonicPolys.drop_column('simon', 'n_squared')
sage: MonicPolys.show('simon')

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
```

(continues on next page)
```python
>>> from sage.all import *
>>> MonicPolys = SQLDatabase()
>>> MonicPolys.create_table('simon', {'n': {'sql': 'INTEGER', 'index': True}})
>>> for n in range(Integer(20)): MonicPolys.add_row('simon', (n,))
>>> MonicPolys.add_column('simon', 'n_squared', {'sql': 'INTEGER'}, Integer(0))
>>> MonicPolys.drop_column('simon', 'n_squared')
>>> MonicPolys.show('simon')

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
</tbody>
</table>
```

**drop_data_from_table** (*table_name*)

Remove all rows from *table_name*.

**EXAMPLES:**

```python
sage: D = SQLDatabase()
sage: D.create_table('simon', {'col1': {'sql': 'INTEGER'}})
sage: D.add_row('simon', (9,))
sage: D.show('simon')
col1
-------------------
9
sage: D.drop_data_from_table('simon')
```
drop_index(table_name, index_name)

Set the column index_name in table table_name to not be an index. See make_index()

EXAMPLES:

```python
sage: MonicPolys = SQLDatabase()
sage: MonicPolys.create_table('simon', {n:{'sql':'INTEGER', 'index':True},
                                         'n2':{sql': 'INTEGER'}})
sage: MonicPolys.drop_index('simon', 'n')
sage: MonicPolys.get_skeleton()
{'simon': {n: {'index': False,
               'primary_key': False,
               'sql': 'INTEGER',
               'unique': False},
           'n2': {'index': False,
                  'primary_key': False,
                  'sql': 'INTEGER',
                  'unique': False}}}
```

drop_primary_key(table_name, col_name)

Set the column col_name in table table_name not to be a primary key.

A primary key is something like an index, but its main purpose is to link different tables together. This allows searches to be executed on multiple tables that represent maybe different data about the same objects.
NOTE:
This function only changes the column to be non-primary, it does not delete it.

EXAMPLES:

```sage
sage: MonicPolys = SQLDatabase()
sage: MonicPolys.create_table('simon', {'n': {'sql':'INTEGER', 'index':True},
       'n2': {'sql': 'INTEGER'}})
sage: MonicPolys.make_primary_key('simon', 'n2')
sage: MonicPolys.drop_primary_key('simon', 'n2')
sage: MonicPolys.get_skeleton()
{'simon': {'n': {'index': True, 'primary_key': False, 'sql': 'INTEGER', 'unique': False},
           'n2': {'index': False, 'primary_key': False, 'sql': 'INTEGER', 'unique': True}}}
```

```python
>>> from sage.all import *

>>> MonicPolys = SQLDatabase()

>>> MonicPolys.create_table('simon', {'n': {'sql': 'INTEGER', 'index':True}, 'n2': {'sql': 'INTEGER'}})

>>> MonicPolys.make_primary_key('simon', 'n2')

>>> MonicPolys.drop_primary_key('simon', 'n2')

>>> MonicPolys.get_skeleton()
{'simon': {'n': {'index': True, 'primary_key': False, 'sql': 'INTEGER', 'unique': False},
           'n2': {'index': False, 'primary_key': False, 'sql': 'INTEGER', 'unique': True}}}
```

drop_table(table_name)
Delete table table_name from database.

INPUT:

- table_name – a string

EXAMPLES:

```sage
sage: D = SQLDatabase()
sage: D.create_table('simon', {'col1': {'sql': 'INTEGER'}})

sage: D.show('simon')
col1

-----------------

sage: D.drop_table('simon')
sage: D.get_skeleton()
{}
```

```python
>>> from sage.all import *

>>> D = SQLDatabase()

>>> D.create_table('simon', {'col1': {'sql': 'INTEGER'}})

>>> D.show('simon')
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

```python
>>> D.drop_table('simon')
>>> D.get_skeleton()
{ }
```

**drop_unique** *(table_name, col_name)*

Set the column `col_name` in table `table_name` not store unique values.

**NOTE:**

This function only removes the requirement for entries in `col_name` to be unique, it does not delete it.

**EXAMPLES:**

```python
sage: MonicPolys = SQLDatabase()
sage: MonicPolys.create_table('simon', {'n': {'sql': 'INTEGER', 'index': True}, 'n2': {'sql': 'INTEGER'}})
sage: MonicPolys.make_unique('simon', 'n2')
sage: MonicPolys.drop_unique('simon', 'n2')
sage: MonicPolys.get_skeleton()
{'simon': {'n': {'index': True, 'primary_key': False, 'sql': 'INTEGER', 'unique': False}, 'n2': {'index': False, 'primary_key': False, 'sql': 'INTEGER', 'unique': False}}}
```

```python
>>> from sage.all import *
>>> MonicPolys = SQLDatabase()
>>> MonicPolys.create_table('simon', {'n': {'sql': 'INTEGER', 'index': True}, 'n2': {'sql': 'INTEGER'}})
>>> MonicPolys.make_unique('simon', 'n2')
>>> MonicPolys.drop_unique('simon', 'n2')
>>> MonicPolys.get_skeleton()
{'simon': {'n': {'index': True, 'primary_key': False, 'sql': 'INTEGER', 'unique': False}, 'n2': {'index': False, 'primary_key': False, 'sql': 'INTEGER', 'unique': False}}}
```

**get_connection** *(ignore_warning=None)*

Return a sqlite connection to the database.

You most likely want `get_cursor()` instead, which is used for executing sqlite commands on the database.

Recommended for more advanced users only.

**EXAMPLES:**

```python
sage: D = SQLDatabase(read_only=True)
sage: con = D.get_connection()
doctest::...: RuntimeWarning: Database is read only, using the connection can...
```

(continues on next page)
In order to mute future warnings, set `self.ignore_warnings` to `True`.

```python
from sage.all import *
D = SQLDatabase(read_only=True)
con = D.get_connection()
doctest:.... RuntimeWarning: Database is read only, using the connection can...
    alter the stored data. Set self.ignore_warnings to True in order to mute...
    future warnings.
con = D.get_connection(True)
D.ignore_warnings = True
con = D.get_connection()
D.ignore_warnings = True
con = D.get_connection()
for n in range(10):
    t = con.execute('CREATE TABLE simon(n INTEGER, n2 INTEGER)')
doctest:....: RuntimeWarning: Database is read only, using the connection can...
    alter the stored data. Set self.ignore_warnings to True in order to mute...
    future warnings.
D.ignore_warnings = True
con = D.get_connection()
for n in range(Integer(10)):
    t = con.execute('INSERT INTO simon VALUES($d,$d)'%(n,n**Integer(2)))
D.show('simon')
```

<table>
<thead>
<tr>
<th>n</th>
<th>n2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>81</td>
</tr>
</tbody>
</table>

---

**get_cursor** *(ignore_warning=None)*

Return a pysqlite cursor for the database connection.

A cursor is an input from which you can execute sqlite commands on the database.

Recommended for more advanced users only.

**EXAMPLES:**
sage: DB = SQLDatabase()
sage: DB.create_table('simon',{'a1':{'sql':'bool'}, 'b2':{'sql':'int'}})
sage: DB.add_row('simon',(0,1))
sage: DB.add_rows('simon',[(0,0),(1,1),(1,2)])
sage: DB.add_rows('simon',[(0,0),(4,0),(5,1)], ['b2','a1'])
sage: cur = DB.get_cursor()
sage: (cur.execute('select * from simon')).fetchall()
[(0, 1), (0, 0), (1, 1), (1, 2), (0, 0), (0, 4), (1, 5)]

>>> from sage.all import *
>>> DB = SQLDatabase()
>>> DB.create_table('simon',{'a1':{'sql':'bool'}, 'b2':{'sql':'int'}})
>>> DB.add_row('simon',(Integer(0),Integer(1)))
>>> DB.add_rows('simon',[(Integer(0),Integer(0)),(Integer(1),Integer(1)),
   (Integer(1),Integer(2))])
>>> DB.add_rows('simon',[(Integer(0),Integer(0)),(Integer(4),Integer(0)),
   (Integer(5),Integer(1))], ['b2','a1'])
>>> cur = DB.get_cursor()
>>> (cur.execute('select * from simon')).fetchall()
[(0, 1), (0, 0), (1, 1), (1, 2), (0, 0), (0, 4), (1, 5)]

get_skeleton (check=False)

Return a dictionary representing the hierarchical structure of the database, in the following format:

| - skeleton -- a triple-indexed dictionary
|   | - outer key -- table name
|   |   | - inner key -- column name
|   |   |   | - inner inner key -- one of the following:
|       |   |       | - primary_key -- boolean, whether column has been set as primary key
|       |   |       | - index -- boolean, whether column has been set as index
|       |   |       | - unique -- boolean, whether column has been set as unique
|       |   |       | - sql -- one of 'TEXT', 'BOOLEAN', 'INTEGER', 'REAL', or other user defined type

For example:

{"table1":{"col1":{"primary_key":False, 'index':True, 'unique': False, 'sql': 'REAL'}}}

INPUT:

- check – if True, checks to make sure the database’s actual structure matches the skeleton on record.

EXAMPLES:

sage: GDB = GraphDatabase()
sage: GDB.get_skeleton()  # slightly random output
{"aut_grp":{"aut_grp_size":{"index": True, 'unique': False, 'primary_key': False, 'sql': 'INTEGER'},
   ...
   "num_vertices":{"index": True, 'unique': False, 'primary_key': False, 'sql': 'INTEGER'}}}
Utilities, Release 10.4

```
>>> from sage.all import *
>>> GDB = GraphDatabase()
>>> GDB.get_skeleton()
{'aut_grp': {'aut_grp_size': {'index': True,
                             'unique': False,
                             'primary_key': False,
                             'sql': 'INTEGER'},
              ...
              'num_vertices': {'index': True,
                              'unique': False,
                              'primary_key': False,
                              'sql': 'INTEGER'}}}
```

**make_index** (*col_name*, *table_name*, **unique=False**)  
Set the column *col_name* in table *table_name* to be an index, that is, a column set up to do quick searches on.

**INPUT:**
- *col_name* – a string
- *table_name* – a string
- **unique** – requires that there are no multiple entries in the column, makes searching faster

**EXAMPLES:**
```
sage: MonicPolys = SQLDatabase()
sage: MonicPolys.create_table('simon', {'n': {'sql': 'INTEGER', 'index': True},
                                          'n2': {'sql': 'INTEGER'}})
sage: MonicPolys.make_index('n2', 'simon')
sage: MonicPolys.get_skeleton()
{'simon': {'n': {'index': True,
                 'primary_key': False,
                 'sql': 'INTEGER',
                 'unique': False},
             'n2': {'index': True,
                    'primary_key': False,
                    'sql': 'INTEGER',
                    'unique': False}}}
```

```
>>> from sage.all import *
>>> MonicPolys = SQLDatabase()
>>> MonicPolys.create_table('simon', {'n': {'sql': 'INTEGER', 'index': True},
                                          'n2': {'sql': 'INTEGER'}})
>>> MonicPolys.make_index('n2', 'simon')
>>> MonicPolys.get_skeleton()
{'simon': {'n': {'index': True,
                 'primary_key': False,
                 'sql': 'INTEGER',
                 'unique': False},
             'n2': {'index': True,
                    'primary_key': False,
                    'sql': 'INTEGER',
                    'unique': False}}}
```

**make_primary_key** (*table_name*, *col*)
Set the column *col* in table *table_name* to be a primary key.
A primary key is something like an index, but its main purpose is to link different tables together. This allows searches to be executed on multiple tables that represent maybe different data about the same objects.

NOTE:

Some SQL features, such as automatically incrementing primary key, require the full word 'INTEGER', not just 'INT'.

EXAMPLES:

```
sage: MonicPolys = SQLDatabase()
sage: MonicPolys.create_table('simon', {'n':{'sql':'INTEGER', 'index':True}, 'n2':{'sql':'INTEGER'}})
sage: MonicPolys.make_primary_key('simon', 'n2')
sage: MonicPolys.get_skeleton()
{'simon': {'n': {'index': True, 'primary_key': False, 'sql': 'INTEGER', 'unique': False}, 'n2': {'index': False, 'primary_key': True, 'sql': 'INTEGER', 'unique': True}}}
```

```
>>> from sage.all import *
>>> MonicPolys = SQLDatabase()
>>> MonicPolys.create_table('simon', {'n':{'sql':'INTEGER', 'index':True}, 'n2':{'sql':'INTEGER'}})
>>> MonicPolys.make_unique('simon', 'n2')
>>> MonicPolys.get_skeleton()
{'simon': {'n': {'index': True, 'primary_key': False, 'sql': 'INTEGER', 'unique': False}, 'n2': {'index': False, 'primary_key': False, 'sql': 'INTEGER', 'unique': True}}}
```

```
make_unique (table_name, col_name)

Set the column col_name in table table_name to store unique values.

NOTE:

This function only adds the requirement for entries in col_name to be unique, it does not change the values.

EXAMPLES:
```
sage: MonicPolys = SQLDatabase()
sage: MonicPolys.create_table('simon', {'n':{'sql':'INTEGER', 'index':True}, 'n2':{'sql':'INTEGER'}})
sage: MonicPolys.make_unique('simon', 'n2')
sage: MonicPolys.get_skeleton()
{'simon': {'n': {'index': True, 'primary_key': False, 'sql': 'INTEGER', 'unique': False}, 'n2': {'index': False, 'primary_key': False, 'sql': 'INTEGER', 'unique': True}}}
```

(continues on next page)
Utilities, Release 10.4

>>> from sage.all import *

>>> MonicPolys = SQLDatabase()

>>> MonicPolys.create_table('simon', {'n':{'sql':'INTEGER', 'index':True}, 'n2'→:{'sql':'INTEGER'}})

>>> MonicPolys.make_unique('simon', 'n2')

>>> MonicPolys.get_skeleton()

{'simon': {'n': {'index': True, 'primary_key': False, 'sql': 'INTEGER', 'unique': False}, 'n2': {'index': False, 'primary_key': False, 'sql': 'INTEGER', 'unique': True}}}

query(*args, **kwds)

Create a SQLQuery on this database.

For full class details, type SQLQuery? and press Shift + Enter.

EXAMPLES:

sage: D = SQLDatabase()

sage: D.create_table('simon', {'wolf':{'sql':'BOOLEAN'}, 'tag':{'sql':'INTEGER'→}})

sage: q = D.query({'table_name':'simon', 'display_cols':['tag'], 'expression'→:{'wolf','=','1'}})

sage: q.get_query_string()

'SELECT simon.tag FROM simon WHERE simon.wolf = ?'

sage: q.__param_tuple__

(1,)

sage: q = D.query(query_string='SELECT tag FROM simon WHERE wolf=?',param_→tuple=(Integer(1),))

sage: q.get_query_string()

'SELECT tag FROM simon WHERE wolf=?'

sage: q.__param_tuple__

(1,)

>>> from sage.all import *

>>> D = SQLDatabase()

>>> D.create_table('simon', {'wolf':{'sql':'BOOLEAN'}, 'tag':{'sql':'INTEGER'}})

>>> q = D.query({'table_name':'simon', 'display_cols':['tag'], 'expression':[→'wolf','=','1']})

>>> q.get_query_string()

'SELECT simon.tag FROM simon WHERE simon.wolf = ?'

>>> q.__param_tuple__

(1,)

>>> q = D.query(query_string='SELECT tag FROM simon WHERE wolf=?',param_→tuple=(Integer(1),))

>>> q.get_query_string()

'SELECT tag FROM simon WHERE wolf=?'

>>> q.__param_tuple__

(1,)
**rename_table** *(table_name, new_name)*

Rename the table `table_name` to `new_name`.

**EXAMPLES:**

```python
sage: D = SQLDatabase()
sage: D.create_table('simon',{col1:{sql:'INTEGER'}})
sage: D.show('simon')
col1
--------------
sage: D.rename_table('simon', 'lucy')
sage: D.show('simon')
Traceback (most recent call last):
...
RuntimeError: Failure to fetch data.
sage: D.show('lucy')
col1
--------------
```

```python
>>> from sage.all import *

D = SQLDatabase()
D.create_table('simon',{col1:{sql:'INTEGER'}})
D.show('simon')
D.rename_table('simon', 'lucy')
D.show('simon')
Traceback (most recent call last):
...
RuntimeError: Failure to fetch data.
D.show('lucy')
col1
--------------
```

**save** *(filename)*

Save the database to the specified location.

**EXAMPLES:**

```python
sage: MonicPolys = SQLDatabase()
sage: MonicPolys.create_table('simon', {'n':{sql:'INTEGER', 'index':True}})
sage: for n in range(20): MonicPolys.add_row('simon', (n,))
sage: import tempfile
sage: with tempfile.TemporaryDirectory() as d:
    dbpath = os.path.join(d, "sage.db")
    MonicPolys.save(dbpath)
    N = SQLDatabase(dbpath)
    N.show('simon')

n
--------------
0
1
2
3
4
5
6
7
```

(continues on next page)
```python
>>> from sage.all import *

>>> MonicPolys = SQLDatabase()

>>> MonicPolys.create_table('simon', {'n': {'sql': 'INTEGER', 'index': True}})

>>> for n in range(Integer(20)): MonicPolys.add_row('simon', (n,))

>>> import tempfile

>>> with tempfile.TemporaryDirectory() as d:
...
  dbpath = os.path.join(d, "sage.db")
...
  MonicPolys.save(dbpath)
...
  N = SQLDatabase(dbpath)
...
  N.show('simon')
```

```
<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

**show** (*table_name*, **kwds**)

Show an entire table from the database.

**EXAMPLES:**

```python
sage: DB = SQLDatabase()
sage: DB.create_table('simon', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
sage: DB.add_data('simon', [(0, 0), (1, 1), (1, 2)])
sage: DB.show('simon')
a1  b2
```
```python
>>> from sage.all import *
>>> DB = SQLDatabase()
>>> DB.create_table('simon', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
>>> DB.add_data('simon', [(Integer(0), Integer(0)), (Integer(1), Integer(1)),
                       (Integer(1), Integer(2))])
>>> DB.show('simon')
a1  b2
0   0
1   1
1   2
```

**vacuum()**

Clean the extra hard disk space used up by a database that has recently shrunk.

**EXAMPLES:**

```python
sage: DB = SQLDatabase()
sage: DB.create_table('simon', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
sage: DB.add_row('simon', (0, 1))
sage: DB.add_data('simon', [(0, 0), (1, 1), (1, 2)])
sage: DB.add_data('simon', [(0, 0), (4, 0), (5, 1)], ['b2', 'a1'])
sage: DB.drop_column('simon', 'b2')
sage: DB.commit()
sage: DB.vacuum()
```

```python
>>> from sage.all import *
>>> DB = SQLDatabase()
>>> DB.create_table('simon', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
>>> DB.add_row('simon', (Integer(0), Integer(1)))
>>> DB.add_data('simon', [(0, 0), (1, 1), (1, 2)])
>>> DB.add_data('simon', [(0, 0), (4, 0), (5, 1)], ['b2', 'a1'])
>>> DB.drop_column('simon', 'b2')
>>> DB.commit()
>>> DB.vacuum()
```

```python
class sage.databases.sql_db.SQLQuery(database, *args, **kwds)

Bases: SageObject

A query for a SQLite database.

**INPUT:**

- database – a SQLDatabase object
- query_dict – a dictionary specifying the query itself. The format is:

```json
{ 'table_name': 'tblname', 'display_cols': [ 'col1', 'col2', 'col3' ], 'expression' => [ 'col1', operator, value] }
```
NOTE:
Every SQL type we are using is ultimately represented as a string, so if you wish to save actual strings to a
database, you actually need to do something like: “value”.

See the documentation of SQLDatabase for an introduction to using SQLite in Sage.

EXAMPLES:

```python
sage: D = SQLDatabase()
sage: D.create_table('simon',{'a1':{'sql':'bool', 'primary_key':False}, 'b2':{'sql':'int'}})
sage: D.add_data('simon',[(0,0),(1,2),(2,4)])
sage: r = SQLQuery(D, {'table_name':'simon', 'display_cols':['a1'], 'expression':[b2,'<=',3]})
sage: r.show()
a1
-------------
0
1
```

Test that Issue #27562 is fixed:

```python
sage: D = SQLDatabase()
sage: r = SQLQuery(D, {'table_name':'simon', 'display_cols':['a1'], 'expression':[b2,'<=',3]})
Traceback (most recent call last):
... ValueError: Database has no table simon
sage: D.create_table('simon',{'a1':{'sql':'bool', 'primary_key':False}, 'b2':{'sql':'int'}})
sage: D.create_table('simon',{'a1':{'sql':'bool', 'primary_key':False}, 'b2':{'sql':'int'}})
Traceback (most recent call last):
... ValueError: Database already has a table named simon
sage: SQLQuery(D, {'table_name':'simon', 'display_cols':['a1'], 'expression':['c1','>','2']})
Traceback (most recent call last):
... ValueError: Table has no column c1
```

```python
>>> from sage.all import *
```
>>> r = SQLQuery(D, {'table_name': 'simon', 'display_cols': ['a1'], 'expression': [{'b2': '<=', Integer(3)}]})
Traceback (most recent call last):
...  
ValueError: Database has no table simon
>>> D.create_table('simon', {'a1': {'sql': 'bool', 'primary_key': False}, 'b2': {'sql': 'int'}})
>>> D.create_table('simon', {'a1': {'sql': 'bool', 'primary_key': False}, 'b2': {'sql': 'int'}})
Traceback (most recent call last):
...  
ValueError: Database already has a table named simon
>>> SQLQuery(D, {'table_name': 'simon', 'display_cols': ['a1'], 'expression': ['c1', '>', Integer(2)]})
Traceback (most recent call last):
...  
ValueError: Table has no column c1

get_query_string()
Return a copy of the query string.

EXAMPLES:

```python
sage: G = GraphDatabase()
sage: q = 'SELECT graph_id,graph6,num_vertices,num_edges FROM graph_data WHERE graph_id<=(?) AND num_vertices=(?)'
sage: param = (22,5)
sage: SQLQuery(G,q,param).get_query_string()
'SELECT graph_id,graph6,num_vertices,num_edges FROM graph_data WHERE graph_id<=(?) AND num_vertices=(?)'
sage: r = 'SELECT graph6 FROM graph_data WHERE num_vertices<=3'
```

```python
>>> from sage.all import *
```

```python
>>> G = GraphDatabase()
>>> q = 'SELECT graph_id,graph6,num_vertices,num_edges FROM graph_data WHERE graph_id<=(?) AND num_vertices=(?)'
>>> param = (Integer(22),Integer(5))
>>> SQLQuery(G,q,param).get_query_string()
'SELECT graph_id,graph6,num_vertices,num_edges FROM graph_data WHERE graph_id<=(?) AND num_vertices=(?)'
>>> r = 'SELECT graph6 FROM graph_data WHERE num_vertices<=3'
```

```python
>>> SQLQuery(G,r).get_query_string()
'SELECT graph6 FROM graph_data WHERE num_vertices<=3'
```

intersect (other, join_table=None, join_dict=None, in_place=False)
Return a new SQLQuery that is the intersection of self and other. join_table and join_dict can be None iff the two queries only search one table in the database. All display columns will be concatenated in order: self display cols + other display cols.

INPUT:

- other – the SQLQuery to intersect with
- join_table – base table to join on (This table should have at least one column in each table to join on).

2.4. Database Access
• **join_dict** – a dictionary that represents the join structure for the new query. (Must include a mapping for all tables, including those previously joined in either query). Structure is given by:

```
{'join_table1': ('corr_base_col1', 'col1'), 'join_table2': ('corr_base_col2', 'col2')}
```

where **join_table1** is to be joined with **join_table** on **join_table.corr_base_col1 = join_table1.col1**

**EXAMPLES:**

```
sage: DB = SQLDatabase()
sage: DB.create_table('simon', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
sage: DB.create_table('lucy', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
sage: DB.add_data('simon', [(0,5),(1,4)])
sage: DB.add_data('lucy', [(1,1),(1,4)])
sage: q = SQLQuery(DB, {'table_name': 'lucy', 'display_cols': ['b2'], 'expression': ['a1', '=', 1]})
sage: r = SQLQuery(DB, {'table_name': 'simon', 'display_cols': ['a1'], 'expression': ['b2', '<=', 6]})
sage: s = q.intersect(r, 'simon', {'lucy': (a1,a1)})
sage: s.get_query_string()
'SELECT lucy.b2, simon.a1 FROM simon INNER JOIN lucy ON simon.a1 = lucy.a1 WHERE ( lucy.a1 = ? ) AND ( simon.b2 <= ? )'
sage: s.query_results()
[(1, 1), (4, 1)]
sage: s = q.intersect(r)
Traceback (most recent call last):
... ValueError: Input queries query different tables but join parameters are NoneType
```

```
>>> from sage.all import *
>>> DB = SQLDatabase()
>>> DB.create_table('simon', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
>>> DB.create_table('lucy', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
>>> DB.add_data('simon', [(Integer(0),Integer(5)),(Integer(1),Integer(4))])
>>> DB.add_data('lucy', [(Integer(1),Integer(1)),(Integer(1),Integer(4))])
>>> q = SQLQuery(DB, {'table_name': 'lucy', 'display_cols': ['b2'], 'expression': ['a1', '=', Integer(1)]})
>>> r = SQLQuery(DB, {'table_name': 'simon', 'display_cols': ['a1'], 'expression': ['b2', '<=', Integer(6)]})
>>> s = q.intersect(r)
>>> s.get_query_string()  # s.__query_string__ == q.__query_string__ True
'SELECT lucy.b2, simon.a1 FROM simon INNER JOIN lucy ON simon.a1 = lucy.a1 WHERE ( lucy.a1 = ? ) AND ( simon.b2 <= ? )'
>>> s.query_results()
[(1, 1), (4, 1)]
>>> s = q.intersect(r)
Traceback (most recent call last):
... ValueError: Input queries query different tables but join parameters are NoneType
```
query_results()

Run the query by executing the __query_string__. Return the results of the query in a list.

EXAMPLES:

```python
sage: G = GraphDatabase()
sage: q = SELECT graph_id, graph6, num_vertices, num_edges FROM graph_data
   WHERE graph_id<= (?) AND num_vertices= (?)
sage: param = (22, 5)
sage: Q = SQLQuery(G, q, param)

>>> Q.query_results()
[(18, 'D??', 5, 0), (19, 'D?C', 5, 1), (20, 'D?K', 5, 2),
 (21, 'D?O', 5, 2), (22, 'D?[', 5, 3)]
```

```python
sage: R = SQLQuery(G, {'table_name': 'graph_data', 'display_cols': ['graph6'],
   'expression': ['num_vertices', '=', Integer(4)]})

>>> R.query_results()
[('C?',), ('C@',), ('CB',), ('CK',), ('CF',), ('CJ',),
 ('CL',), ('CN',), ('C[',), ('C^',), ('C~',)]
```

show(**kwds)

Display the result of the query in table format.

KEYWORDS:

- max_field_size – how wide each field can be
- format_cols – a dictionary that allows the user to specify the format of a column’s output by supplying a function. The format of the dictionary is:

  ```python
  {'column_name': (lambda x: format_function(x))}
  ```

- plot_cols – a dictionary that allows the user to specify that a plot should be drawn by the object generated by a data slice. Note that plot kwds are permitted. The dictionary format is:

  ```python
  {'column_name': (lambda x: plot_function(x)), **kwds}
  ```

- relabel_cols – a dictionary to specify a relabeling of column headers. The dictionary format is:
Utilities, Release 10.4

• id_col – reference to a column that can be used as an object identifier for each row

EXAMPLES:

```python
db = SQLDatabase()
db.create_table('simon', {'a1': {'sql': 'bool', 'primary_key': False}, 'b2': {'sql': 'int'}})
db.add_data('simon', [(0,0), (1,1), (1,2)])
r = SQLQuery(db, {'table_name': 'simon', 'display_cols': ['a1'], 'expression': ['b2', '<=', 6]})
r.show()
a1
-------------------
0
1

>>> from sage.all import *
>>> db = SQLDatabase()
>>> db.create_table('simon', {'a1': {'sql': 'bool', 'primary_key': False}, 'b2': {'sql': 'int'}})
>>> db.add_data('simon', [(0,Integer(0),Integer(0)), (1,Integer(1),Integer(1)), (1,Integer(1),Integer(2))])
>>> r = SQLQuery(db, {'table_name': 'simon', 'display_cols': ['a1'], 'expression': ['b2', '<=', 6]})
>>> r.show()
a1
-------------------
0
1
```

(continues on next page)
>>> from sage.graphs.graph_database import valid_kwds, data_to_degseq
>>> relabel = {}
>>> for col in valid_kwds:
...   relabel[col] = '_'.join([word.capitalize() for word in col.split('_')])
>>> q = GraphQuery(display_cols=['graph6', 'degree_sequence'], num_vertices=Integer(4))
>>> SQLQuery.show(q, format_cols={'degree_sequence':(lambda x,y: data_to_degseq(x,y))}, relabel_cols=relabel, id_col='graph6')

```
<table>
<thead>
<tr>
<th>Graph6</th>
<th>Degree Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>[0, 0, 0, 0]</td>
</tr>
<tr>
<td>C@</td>
<td>[0, 0, 1, 1]</td>
</tr>
<tr>
<td>CB</td>
<td>[0, 1, 1, 2]</td>
</tr>
<tr>
<td>CF</td>
<td>[1, 1, 1, 3]</td>
</tr>
<tr>
<td>CJ</td>
<td>[0, 2, 2, 2]</td>
</tr>
<tr>
<td>CK</td>
<td>[1, 1, 1, 1]</td>
</tr>
<tr>
<td>CL</td>
<td>[1, 1, 2, 2]</td>
</tr>
<tr>
<td>CN</td>
<td>[1, 2, 2, 3]</td>
</tr>
<tr>
<td>C]</td>
<td>[2, 2, 2, 2]</td>
</tr>
<tr>
<td>C^</td>
<td>[2, 2, 3, 3]</td>
</tr>
<tr>
<td>C~</td>
<td>[3, 3, 3, 3]</td>
</tr>
</tbody>
</table>
```

**union** *(other, join_table=None, join_dict=None, in_place=False)*

Return a new SQLQuery that is the union of self and other. *join_table* and *join_dict* can be None if the two queries only search one table in the database. All display columns will be concatenated in order: self display cols + other display cols.

**INPUT:**

- **other** – the SQLQuery to union with
- **join_table** – base table to join on (This table should have at least one column in each table to join on).
- **join_dict** – a dictionary that represents the join structure for the new query. (Must include a mapping for all tables, including those previously joined in either query). Structure is given by:

\[
\{\text{join_table1}: (\text{corr_base_col1}, \text{col1}), \text{join_table2}: (\text{corr_base_col2}, \text{col2})\}
\]

where *join_table1* is to be joined with *join_table* on *join_table1*. corr_base_col1=join_table1.col1

**EXAMPLES:**

```sage
sage: DB = SQLDatabase()
sage: DB.create_table('simon', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
sage: DB.create_table('lucy', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
sage: DB.add_data('simon', [(0,5),(1,4)])
sage: DB.add_data('lucy', [(1,1),(1,4)])
sage: q = SQLQuery(DB, {'table_name': 'lucy', 'display_cols': ['b2'], 'expression': ['a1', '=', 1]})
sage: r = SQLQuery(DB, {'table_name': 'simon', 'display_cols': ['a1'], 'expression': ['b2', '<=', 6]})
sage: s = q.union(r, 'simon', {'lucy':('a1','a1')})
sage: s.get_query_string()
'SELECT lucy.b2,simon.a1 FROM simon INNER JOIN lucy ON
```

(continues on next page)
Utilities, Release 10.4

```python
sage: s.query_results()
[(1, 1), (4, 1)]
```

```python
>>> from sage.all import *
>>> DB = SQLDatabase()
>>> DB.create_table('simon', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
>>> DB.create_table('lucy', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
>>> DB.add_data('simon', [(Integer(0), Integer(5)), (Integer(1), Integer(4))])
>>> DB.add_data('lucy', [(Integer(1), Integer(1)), (Integer(1), Integer(4))])
>>> q = SQLQuery(DB, {'table_name': 'lucy', 'display_cols': ['b2'], 'expression ->': ['a1', '=', Integer(1)]})
>>> r = SQLQuery(DB, {'table_name': 'simon', 'display_cols': ['a1'], 'expression ->': ['b2', '<=', Integer(6)]})
>>> s = q.union(r, simon, {lucy: (a1, a1)})
>>> s.get_query_string()
SELECT lucy.b2, simon.a1 FROM simon INNER JOIN lucy ON simon.a1=lucy.a1 WHERE ( lucy.a1 = ? ) OR ( simon.b2 <= ? )
```

```python
sage.databases.sql_db.construct_skeleton(database)

Construct a database skeleton from the SQL data. The skeleton data structure is a triple indexed dictionary of the following format:

- skeleton -- a triple-indexed dictionary
- outer key -- table name
- inner key -- column name
- inner inner key -- one of the following:
  - 'primary_key' -- boolean, whether column has been set as primary key
  - 'index' -- boolean, whether column has been set as index
  - 'unique' -- boolean, whether column has been set as unique
  - 'sql' -- one of 'TEXT', 'BOOLEAN', 'INTEGER', 'REAL', or other user defined type

An example skeleton of a database with one table, that table with one column:

```
{'table1': {'col1': {'primary_key': False, 'unique': True, 'index': True, 'sql': 'REAL ->'}}}
```

EXAMPLES:

```python
sage: G = SQLDatabase(GraphDatabase().__dblocation__, False)
sage: from sage.databases.sql_db import construct_skeleton
sage: sorted(construct_skeleton(G))
['aut_grp', 'degrees', 'graph_data', 'misc', 'spectrum']
```

```python
>>> from sage.all import *
>>> G = SQLDatabase(GraphDatabase().__dblocation__, False)
>>> from sage.databases.sql_db import construct_skeleton
>>> sorted(construct_skeleton(G))
['aut_grp', 'degrees', 'graph_data', 'misc', 'spectrum']
```

sage.databases.sql_db.regex(expr, item)

Function to define regular expressions in pysqlite.
OUTPUT:

- True if parameter item matches the regular expression parameter expr
- False otherwise (i.e.: no match)

REFERENCES:

- [Ha2005]

EXAMPLES:

```python
sage: from sage.databases.sql_db import regexp
sage: regexp('.s.*', 'cs')
True
sage: regexp('.s.*', 'ccs')
False
sage: regexp('.s.*', 'cscccc')
True
```

```python
>>> from sage.all import *
>>> from sage.databases.sql_db import regexp

>>> regexp('.s.*', 'cs')
True

>>> regexp('.s.*', 'ccs')
False

>>> regexp('.s.*', 'cscccc')
True
```

sage.databases.sql_db.verify_column(col_dict)

Verify that col_dict is in proper format, and return a dict with default values filled in. Proper format:

```python
{\'primary_key\':False, \'index\':False, \'unique\':False, \'sql\':'REAL'}
```

EXAMPLES:

```python
sage: from sage.databases.sql_db import verify_column
sage: col = {\'sql\':'BOOLEAN'}
sage: verify_column(col)
{\'index\':False, \'primary_key\':False, \'sql\':'BOOLEAN', \'unique\':False}

sage: col = {\'primary_key\':True, \'sql\':'INTEGER'}

sage: verify_column(col)
{\'index\':False, \'primary_key\':True, \'sql\':'INTEGER', \'unique\':True}

sage: verify_column({})
Traceback (most recent call last):
...
ValueError: SQL type must be declared, e.g. {\'sql\':'REAL'}.
```

```python
>>> from sage.all import *
>>> from sage.databases.sql_db import verify_column

>>> col = {\'sql\':'BOOLEAN'}

>>> verify_column(col)
{\'index\':False, \'primary_key\':False, \'sql\':'BOOLEAN', \'unique\':False}

>>> col = {\'primary_key\':True, \'sql\':'INTEGER'}

>>> verify_column(col)
{\'index\':True, \'primary_key\':True, \'sql\':'INTEGER', \'unique\':True}

>>> verify_column({})
Traceback (most recent call last):
...
```

(continues on next page)
ValueError: SQL type must be declared, e.g. ('sql': 'REAL').

sage.databases.sql_db.verify_operator(\texttt{operator})

Check that \texttt{operator} is one of the allowed strings. Legal operators include the following strings:

\begin{itemize}
  \item '='
  \item '<='
  \item '>='
  \item '<'
  \item '>
  \item '<>'
  \item 'like'
  \item 'regexp'
  \item 'is null'
  \item 'is not null'
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.databases.sql_db import verify_operator
sage: verify_operator('<=')
True
sage: verify_operator('regexp')
True
sage: verify_operator('is null')
True
sage: verify_operator('not_an_operator')
Traceback (most recent call last):
...
TypeError: not_an_operator is not a legal operator.
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> from sage.databases.sql_db import verify_operator
>>> verify_operator('<=')
True
>>> verify_operator('regexp')
True
>>> verify_operator('is null')
True
>>> verify_operator('not_an_operator')
Traceback (most recent call last):
...
TypeError: not_an_operator is not a legal operator.
\end{verbatim}

sage.databases.sql_db.verify_type(\texttt{type})

Verify that the specified \texttt{type} is one of the allowed strings.

\textbf{EXAMPLES:}
```python
sage: from sage.databases.sql_db import verify_type
sage: verify_type('INT')
True
sage: verify_type('int')
True
sage: verify_type('float')
Traceback (most recent call last):
  ...TypeError: float is not a legal type.
```

```python
>>> from sage.all import *
>>> from sage.databases.sql_db import verify_type
>>> verify_type('INT')
True
>>> verify_type('int')
True
>>> verify_type('float')
Traceback (most recent call last):
  ...TypeError: float is not a legal type.
```

### 2.5 Warnings

#### 2.5.1 Stopgaps

```python
exception sage.misc.stopgap.StopgapWarning

Bases: Warning

This class is used to warn users of a known issue that may produce mathematically incorrect results.
```

```python
sage.misc.stopgap.set_state(mode)

Enable or disable the stopgap warnings.
```

**INPUT:**

- `mode` – (bool); if True, enable stopgaps; otherwise, disable.

**EXAMPLES:**

```python
sage: import sage.misc.stopgap
sage: sage.misc.stopgap.set_state(False)
>>> sage.misc.stopgap.stopgap("Displays nothing.", 12345)
...StopgapWarning: Displays nothing.
This issue is being tracked at https://github.com/sagemath/sage/issues/123456.
>>> sage.misc.stopgap.set_state(True)
```

```python
>>> from sage.all import *
>>> import sage.misc.stopgap
```

```python
>>> sage.misc.stopgap.set_state(False)
```

```python
>>> sage.misc.stopgap.stopgap("Displays nothing.", Integer(12345))
```

(continues on next page)
sage.misc.stopgap.stopgap(message, issue_no)

Issue a stopgap warning.

INPUT:

• message – an explanation of how an incorrect answer might be produced.

• issue_no – an integer, giving the number of the Github issue tracking the underlying issue.

EXAMPLES:

```python
sage: import sage.misc.stopgap
sage: sage.misc.stopgap.set_state(True)
sage: sage.misc.stopgap.stopgap("Computation of heights on elliptic curves over →number fields can return very imprecise results.", 12509)
doctest:...
...StopgapWarning: Computation of heights on elliptic curves over number fields→can return very imprecise results.
This issue is being tracked at https://github.com/sagemath/sage/issues/12509.
sage: sage.misc.stopgap.stopgap("This is not printed", 12509)
sage: sage.misc.stopgap.set_state(False)  # so rest of doctesting fine
```

```python
>>> from sage.all import *
>>> import sage.misc.stopgap

>>> sage.misc.stopgap.set_state(True)

>>> sage.misc.stopgap.stopgap("Computation of heights on elliptic curves over →number fields can return very imprecise results.", Integer(12509))
doctest:...
...StopgapWarning: Computation of heights on elliptic curves over number fields→can return very imprecise results.
This issue is being tracked at https://github.com/sagemath/sage/issues/12509.
>>> sage.misc.stopgap.stopgap("This is not printed", Integer(12509))
>>> sage.misc.stopgap.set_state(False)  # so rest of doctesting fine
```
2.5.2 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 github issue number Issue #13109, which is where this mandatory argument to `deprecation()` was introduced.

Functions and classes

class sage.misc.superseded.DeprecatedFunctionAlias (issue_number, func, module, instance=None, unbound=None)

Bases: object

A wrapper around methods or functions which automatically prints a 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_function_alias (issue_number, func)

Create an aliased version of a function or a method which raises a deprecation warning message.

If f is a function or a method, write `g = deprecated_function_alias(issue_number, f)` to make a deprecated aliased version of f.

INPUT:

• `issue_number` – integer. The github issue number where the deprecation is introduced.

• `func` – the function or method to be aliased

EXAMPLES:

```python
sage: from sage.misc.superseded import deprecated_function_alias
sage: g = deprecated_function_alias(13109, number_of_partitions)  # needs sage.combinat sage.libs.flint
doctest:...: DeprecationWarning: g is deprecated. Please use sage.combinat.partition.number_of_partitions instead.
See https://github.com/sagemath/sage/issues/13109 for details.
```

This also works for methods:
sage: class cls:
    ....:    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 https://github.com/sagemath/sage/issues/13109 for details.
42

>>> from sage.all import *
>>> class cls:
...    def new_meth(self):
...        return Integer(42)
...    old_meth = deprecated_function_alias(Integer(13109), new_meth)
>>> cls().old_meth()
doctest:....: DeprecationWarning: old_meth is deprecated. Please use new_meth...
→ instead.
See https://github.com/sagemath/sage/issues/13109 for details.
42

Issue #11585:

sage: def a():
    pass
sage: b = deprecated_function_alias(13109, a)
sage: b()
doctest:....: DeprecationWarning: b is deprecated. Please use a instead.
See https://github.com/sagemath/sage/issues/13109 for details.

>>> from sage.all import *
>>> def a():
...    pass
...    b = deprecated_function_alias(Integer(13109), a)
...    b()
>>> b()
doctest:....: DeprecationWarning: b is deprecated. Please use a instead.
See https://github.com/sagemath/sage/issues/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(issue_number, message, stacklevel=4)

Issue a deprecation warning.

INPUT:
  • issue_number – integer. The github issue number where the deprecation is introduced.
  • message – string. An explanation why things are deprecated and by what it should be replaced.
  • stack_level – (default: 4) an integer. This is passed on to warnings.warn().

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 https://github.com/sagemath/sage/issues/13109 for details.
```python
>>> from sage.all import *
>>> def foo():
...     sage.misc.superseded.deprecation(Integer(13109), 'the function foo is replaced by bar')
>>> foo()

doctest:...: DeprecationWarning: the function foo is replaced by bar
See https://github.com/sagemath/sage/issues/13109 for details.
```

See also:

- `experimental()`,
- `warning()`,
- `sage.misc.superseded.deprecation_cython(issue_number, message, stacklevel=3)`

Issue a deprecation warning – for use in cython functions

```python
class sage.misc.superseded.experimental(issue_number, stacklevel=4)
```

Bases: object

A decorator which warns about the experimental/unstable status of the decorated class/method/function.

**INPUT:**

- `issue_number` – an integer. The github issue number where this code was introduced.
- `stack_level` – (default: 4) an integer. This is passed on to `warnings.warn()`.

**EXAMPLES:**

```python
sage: @sage.misc.superseded.experimental(issue_number=79997)
...     def foo(*args, **kwargs):
...         print("{} {}\n".format(args, kwargs))
sage: foo(7, what='Hello')
doctest:...: FutureWarning: This class/method/function is marked as experimental. It, its functionality or its interface might change without a formal deprecation.
See https://github.com/sagemath/sage/issues/79997 for details.
(7,) {'what': 'Hello'}
```

```python
>>> from sage.all import *
>>> @sage.misc.superseded.experimental(issue_number=Integer(79997))
... def foo(*args, **kwargs):
...     print("{} {}\n".format(args, kwargs))
>>> foo(Integer(7), what='Hello')

doctest:...: FutureWarning: This class/method/function is marked as experimental. It, its functionality or its interface might change without a formal deprecation.
See https://github.com/sagemath/sage/issues/79997 for details.
(7,) {'what': 'Hello'}
```

```python
sage: class bird(SageObject):
    ...     @sage.misc.superseded.experimental(issue_number=99999)
    ...     def __init__(self, *args, **kwargs):
    ...         print("piep {} {}\n".format(args, kwargs))
sage: _ = bird(99)

doctest:...: FutureWarning: This class/method/function is marked as experimental. It, its functionality or its interface might change without a formal deprecation.
See https://github.com/sagemath/sage/issues/99999 for details.
piep (99,)
```
>>> from sage.all import *
>>> class bird(SageObject):
...     @sage.misc.superseded.experimental(issue_number=Integer(99999))
...     def __init__(self, *args, **kwargs):
...         print("piep {} {}\n".format(args, kwargs))
>>> _ = bird(Integer(99))

doctest:...: FutureWarning: This class/method/function is marked as experimental. It, its functionality or its interface might change without a formal deprecation.
See https://github.com/sagemath/sage/issues/99999 for details.
piep (99,) {}

See also:

experimental(), warning(), deprecation().

sage.misc.superseded.experimental_warning(issue_number, message, stacklevel=4)
Issue a warning that the functionality or class is experimental and might change in future.

INPUT:

- issue_number – an integer. The github issue number where the experimental functionality was introduced.
- message – a string. An explanation what is going on.
- stack_level – (default: 4) an integer. This is passed on to warnings.warn().

EXAMPLES:

sage: def foo():
...     sage.misc.superseded.experimental_warning(66666, 'This function is experimental and might change in future."
sage: foo()
doctest:...: FutureWarning: This function is experimental and might change in future.
See https://github.com/sagemath/sage/issues/66666 for details.

>>> from sage.all import *
>>> def Foo():
...     sage.misc.superseded.experimental_warning(Integer(66666), 'This function is experimental and might change in future.')
>>> Foo()
doctest:...: FutureWarning: This function is experimental and might change in future.
See https://github.com/sagemath/sage/issues/66666 for details.

See also:

mark_as_experimental, warning(), deprecation().

sage.misc.superseded.warning(issue_number, message, warning_class=<class 'Warning'>, stacklevel=3)
Issue a warning.

INPUT:

- issue_number – integer. The github issue number where the deprecation is introduced.
- message – string. An explanation what is going on.
• `warning_class` – (default: `Warning`) a class inherited from a Python `Warning`.

• `stack_level` – (default: 3) an integer. This is passed on to `warnings.warn()`.

EXAMPLES:

```python
sage: def foo():
    ...:     sage.misc.superseded.warning(99999,
    ...:         'The syntax will change in future.',
    ...:         FutureWarning)

sage: foo()
```

```
doctest:...: FutureWarning: The syntax will change in future.
See https://github.com/sagemath/sage/issues/99999 for details.
```

```python
>>> from sage.all import *

>>> def foo():
...     sage.misc.superseded.warning(Integer(99999),
...         'The syntax will change in future.',
...         FutureWarning)

>>> foo()
```

```
doctest:...: FutureWarning: The syntax will change in future.
See https://github.com/sagemath/sage/issues/99999 for details.
```

See also:

`deprecation()`, `experimental()`, `exceptions.Warning`.

## 2.6 Miscellaneous Useful Functions

### 2.6.1 Miscellaneous functions

AUTHORS:

• William Stein

• William Stein (2006-04-26): added workaround for Windows where most users' home directory has a space in it.


```python
class sage.misc.misc.BackslashOperator
    Bases: object

    Implements Matlab-style backslash operator for solving systems:

    \[ \text{A} \backslash \text{b} \]

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

EXAMPLES:

```
sage: prepare("A \\ matrix(QQ,2,1,[1/3,2/3])")
"A * BackslashOperator() * matrix(QQ,Integer(2),Integer(1),[Integer(1)/
\rightarrow Integer(3),'2/3']]")
sage: prepare("A \\ matrix(QQ,2,1,[1/3,2*3])")
'A * BackslashOperator() * matrix(QQ,Integer(2),Integer(1),[Integer(1)/
\rightarrow Integer(3),Integer(2)*Integer(3)])'
```

(continues on next page)
Utilities, Release 10.4

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'

>>> from sage.all import *

>>> 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.misc.misc.compose(f, g)

Return the composition of one-variable functions: f o g

See also 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')
# needs sage.symbolic
sage: h1(x)
# needs sage.symbolic
3*x + 1
sage: h2(x)
# needs sage.symbolic
3*x + 3

>>> from sage.all import *

>>> def g(x): return Integer(3)*x
>>> def f(x): return x + Integer(1)
>>> h1 = compose(f, g)
>>> h2 = compose(g, f)
>>> _ = var('x')     # needs sage.symbolic
>>> h1(x)
# needs sage.symbolic
3*x + 1
>>> h2(x)
# needs sage.symbolic
3*x + 3

sage: _ = function('f g')     # needs sage.symbolic
sage: _ = var('x')     # needs sage.symbolic
sage: compose(f, g)(x)     # needs sage.symbolic
f(g(x))

>>> from sage.all import *
>>> _ = function('f g')     # needs sage.symbolic
>>> _ = var('x')     # needs sage.symbolic
>>> compose(f, g)(x)     # needs sage.symbolic
f(g(x))

sage.misc.misc.exactly_one_is_true(iterable)
Return whether exactly one element of iterable evaluates True.

INPUT:
• iterable – an iterable object

OUTPUT:
A boolean.

Note: The implementation is suggested by stackoverflow entry.

EXAMPLES:

sage: from sage.misc.misc import exactly_one_is_true
sage: exactly_one_is_true([]) False
sage: exactly_one_is_true([True]) True
sage: exactly_one_is_true([False]) False
sage: exactly_one_is_true([True, True]) False
sage: exactly_one_is_true([False, True]) True
sage: exactly_one_is_true([True, False, True]) False

(continues on next page)
Utilities, Release 10.4

sage: exactly_one_is_true([False, True, False])
True

>>> from sage.all import *
>>> from sage.misc.misc import exactly_one_is_true
>>> exactly_one_is_true([])
False
>>> exactly_one_is_true([True])
True
>>> exactly_one_is_true([False])
False
>>> exactly_one_is_true([True, True])
False
>>> exactly_one_is_true([False, True])
True
>>> exactly_one_is_true([True, False, True])
False
>>> exactly_one_is_true([False, True, False])
True

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)

>>> from sage.all import *
>>> cubes = [t**Integer(3) for t in range(-Integer(10),Integer(11))]
>>> exists([(x,y) for x in cubes for y in cubes], lambda v : v[Integer(0)]+v[Integer(1)] == Integer(218))
(True, (-125, 343))
>>> exists([(x,y) for x in cubes for y in cubes], lambda v : v[Integer(0)]+v[Integer(1)] == Integer(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
def f():
    G = get_main_globals()
    assert G['bli'] == 14
    assert G['blo'] == 42
sage: bli = 14
sage: f()
42
```

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'] == Integer(14)
    assert G['blo'] == Integer(42)
>>> bli = Integer(14)
(continues on next page)```
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.

\[ \text{sage.misc.misc.} \text{increase} \text{._} \text{recursion} \text{._} \text{limit} (*\text{args}, **\text{kwds}) \]

Context manager to temporarily change the Python maximum recursion depth.

INPUT:

\[ \cdot \text{increment: increment to add to the current limit} \]

EXAMPLES:

\[
\begin{align*}
\text{sage: from sage.misc.misc import increase_recursion_limit} \\
\text{sage: def rec(n): None if n == 0 else rec(n-1)} \\
\text{sage: rec(10000)} \\
\text{Traceback (most recent call last):} \\
\quad ... \\
\text{RecursionError: maximum recursion depth exceeded...} \\
\text{sage: with increase_recursion_limit(10000): rec(10000)} \\
\text{sage: rec(10000)} \\
\text{Traceback (most recent call last):} \\
\quad ... \\
\text{RecursionError: maximum recursion depth exceeded...}
\end{align*}
\]

\[
\begin{align*}
\text{>>> from sage.all import *} \\
\text{>>> from sage.misc.misc import increase_recursion_limit} \\
\text{>>> def rec(n): None if n == Integer(0) else rec(n-Integer(1))} \\
\text{>>> rec(Integer(10000))} \\
\text{Traceback (most recent call last):} \\
\quad ... \\
\text{RecursionError: maximum recursion depth exceeded...} \\
\text{>>> with increase_recursion_limit(Integer(10000)): rec(Integer(10000))} \\
\text{>>> rec(Integer(10000))} \\
\text{Traceback (most recent call last):} \\
\quad ... \\
\text{RecursionError: maximum recursion depth exceeded...}
\end{align*}
\]

\[ \text{sage.misc.misc.} \text{inject} \text{._} \text{variable} (\text{name, value, warn=True}) \]

Inject a variable into the main global namespace.

INPUT:

\[ \cdot \text{name – a string} \]
\[ \cdot \text{value – anything} \]
\[ \cdot \text{warn – a boolean (default: False)} \]

EXAMPLES:
Utilities, Release 10.4

```python
sage: from sage.misc.misc import inject_variable
sage: inject_variable("a", 314)
sage: a
314

sage: from sage.all import *

sage: from sage.misc.misc import inject_variable

sage: inject_variable("a", Integer(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")

Warnings can be disabled:

sage: b = 3
sage: inject_variable("b", 42, warn=False)
sage: b
42

sage: b = Integer(3)

sage: inject_variable("b", Integer(42), warn=False)

sage: b
42

Use with care!
```
sage.misc.misc.inject_variable_test(name, value, depth)

A function for testing deep calls to inject_variable

EXAMPLES:

```python
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
```

```python
>>> from sage.all import *
>>> from sage.misc.misc import inject_variable_test
>>> inject_variable_test("a0", Integer(314), Integer(0))
```

```python
>>> from sage.all import *
>>> from sage.misc.misc import inject_variable_test
>>> inject_variable_test("a0", Integer(314), Integer(0))
```

sage.misc.misc.is_in_string(line, pos)

Return True if the character at position pos in line occurs within a string.

EXAMPLES:

```python
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
```

```python
>>> from sage.all import *
>>> from sage.misc.misc import is_in_string
>>> line = 'test(\'#\')'
>>> is_in_string(line, line.rfind('#'))
True
>>> 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 test if `it` is an iterator. This is not quite correct since `it` could have a `next` method with a different semantic.

**EXAMPLES:**

```python
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: 'wrong' object is not iterable

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)  # needs sage.combinat
sage: is_iterator(P)  # needs sage.combinat
False
sage: is_iterator(iter(P))  # needs sage.combinat
True
```

```bash
>>> from sage.all import *
>>> it = iter([Integer(1),Integer(2),Integer(3)])
>>> is_iterator(it)
True

>>> class wrong():
...     def __init__(self): self.n = Integer(5)
...     def __next__(self):
...         self.n -= Integer(1)
...         if self.n == Integer(0): raise StopIteration
...         return self.n
>>> x = wrong()
>>> is_iterator(x)
False
>>> list(x)
Traceback (most recent call last):
  ... TypeError: 'wrong' object is not iterable
```
>>> class good(wrong):
...     def __iter__(self):
...         return self
>>> x = good()
>>> is_iterator(x)
True
>>> list(x)
[4, 3, 2, 1]

>>> P = Partitions(Integer(3))  # needs sage.combinat
>>> is_iterator(P)  # needs sage.combinat
False
>>> is_iterator(iter(P))  # needs sage.combinat
True

sage.misc.misc.is_sublist(X, Y)
Test whether X is a sublist of Y.

EXAMPLES:

```python
sage: from sage.misc.misc import is_sublist
sage: S = [1, 7, 3, 4, 18]
sage: is_sublist([1, 7], S)
True
sage: is_sublist([1, 3, 4], S)
True
sage: is_sublist([1, 4, 3], S)
False
sage: is_sublist(S, S)
True
```
OUTPUT:

\[ f(f(...f(x)...)), \text{ where the composition occurs } n \text{ times} \]

EXAMPLES:

```python
sage: def f(x): return x^2 + 1
sage: x = var('x') #
→ needs sage.symbolic
sage: nest(f, 3, x) #
→ needs sage.symbolic
((x^2 + 1)^2 + 1)^2 + 1
```

```python
>>> from sage.all import *
>>> def f(x): return x**Integer(2) + Integer(1)
>>> x = var('x') #
→ needs sage.symbolic
>>> nest(f, Integer(3), x) ...
→ # needs sage.symbolic
((x^2 + 1)^2 + 1)^2 + 1
```

```python
sage: _ = function('f') #
→ needs sage.symbolic
sage: _ = var('x') #
→ needs sage.symbolic
sage: nest(f, 10, x) #
→ needs sage.symbolic
f(f(f(f(f(f(f(f(f(f(x)))))))))
```

```python
>>> from sage.all import *
>>> _ = function('f') #
→ needs sage.symbolic
>>> _ = var('x') #
→ needs sage.symbolic
>>> nest(f, Integer(10), x) ...
→ # needs sage.symbolic
f(f(f(f(f(f(f(f(f(f(x)))))))))))
```

```python
sage: _ = function('f') #
→ needs sage.symbolic
sage: _ = var('x') #
→ needs sage.symbolic
sage: nest(f, 0, x) #
→ needs sage.symbolic
x
```

```python
>>> from sage.all import *
>>> _ = function('f') #
→ needs sage.symbolic
>>> _ = var('x') #
→ needs sage.symbolic
>>> nest(f, Integer(0), x) ...
→ # needs sage.symbolic
x
```

\[ \text{sage.misc.misc.newton_method_sizes}(N) \]

Return a sequence of integers \( 1 = a_1 \leq a_2 \leq \cdots \leq a_n = N \) 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:**

```python
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]
```

>>> from sage.all import *

```python
sage: newton_method_sizes(Integer(17))
[1, 2, 3, 5, 9, 17]
sage: newton_method_sizes(Integer(16))
[1, 2, 4, 8, 16]
sage: newton_method_sizes(Integer(1))
[1]
```

**AUTHORS:**

- David Harvey (2006-09-09)

---

**sage.misc.misc.pad_zeros(s, size=3)**

**EXAMPLES:**

```python
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'
```

>>> from sage.all import *

```python
sage: pad_zeros(Integer(100))
'100'
sage: pad_zeros(Integer(10))
'010'
sage: pad_zeros(Integer(10), Integer(5))
'00010'
sage: pad_zeros(Integer(389), Integer(5))
'00389'
sage: pad_zeros(Integer(389), Integer(10))
'0000000389'
```

**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:**

---

**2.6. Miscellaneous Useful Functions**
• X — list
• s – floating point number between 0 and 1

OUTPUT: list

EXAMPLES:

```python
sage: from sage.misc.misc import is_sublist
sage: S = [1, 7, 3, 4, 18]
```
```python
delist = random_sublist(S, 0.5); sublist  # random
[1, 3, 4]
```
```python
sage: is_sublist(sublist, S)
True
```
```python
delist = random_sublist(S, 0.5); sublist  # random
[1, 3]
```
```python
sage: is_sublist(sublist, S)
True
```
```python
>>> from sage.all import *
>>> from sage.misc.misc import is_sublist
>>> S = [Integer(1), Integer(7), Integer(3), Integer(4), Integer(18)]
```n```python
>>> sublist = random_sublist(S, RealNumber('0.5')); sublist  # random
[1, 3, 4]
```
```python
>>> is_sublist(sublist, S)
True
```
```python
>>> sublist = random_sublist(S, RealNumber('0.5')); sublist  # random
[1, 3]
```
```python
>>> is_sublist(sublist, S)
True
```

```
sage.misc.misc.run_once(func)
Runs a function (successfully) only once.
The running can be reset by setting the has_run attribute to False
```

```
sage.misc.misc.some_tuples(elements, repeat, bound, max_samples=None)
Return an iterator over at most bound number of repeat-tuples of elements.
```

INPUT:
• elements — an iterable
• repeat — integer (default None), the length of the tuples to be returned. If None, just returns entries from elements.
• bound — the maximum number of tuples returned (ignored if max_samples given)
• max_samples — non-negative integer (default None). If given, then a sample of the possible tuples will be returned, instead of the first few in the standard order.

OUTPUT:
If max_samples is not provided, an iterator over the first bound tuples of length repeat, in the standard nested-for-loop order.
If max_samples is provided, a list of at most max_samples tuples, sampled uniformly from the possibilities. In this case, elements must be finite.

```
sage.misc.misc.strunc(s, n=60)
Truncate at first space after position n, adding ‘...’ if nontrivial truncation.
```
sage.misc.misc.try_read(obj, splitlines=False)

Determine if a given object is a readable file-like object and if so read and return its contents.

That is, the object has a callable method named read() which takes no arguments (except self) then the method is executed and the contents are returned.

Alternatively, if the splitlines=True is given, first splitlines() is tried, then if that fails read(). splitlines().

If either method fails, None is returned.

INPUT:

• obj – typically a file or io.BaseIO object, but any other object with a read() method is accepted.

• splitlines – bool, optional; if True, return a list of lines instead of a string.

EXAMPLES:

```python
sage: import io
sage: filename = tmp_filename()
sage: from sage.misc.misc import try_read
sage: with open(filename, 'w') as fobj:
    ....:    _ = fobj.write('a
b
c')
sage: with open(filename) as fobj:
    ....:    print(try_read(fobj))
a
b
c
sage: with open(filename) as fobj:
    ....:    try_read(fobj, splitlines=True)
['a\n', 'b\n', 'c']
```

The following example is identical to the above example on Python 3, but different on Python 2 where open != io.open:

```python
>>> from sage.all import *
>>> import io
>>> filename = tmp_filename()
>>> from sage.misc.misc import try_read
>>> with open(filename, 'w') as fobj:
    ...    _ = fobj.write('a\nb\nc')
>>> with open(filename) as fobj:
    ...    try_read(fobj, splitlines=True)
['a\n', 'b\n', 'c']
```

(continues on next page)
...    print(try_read(fobj))
a
b
c

I/O buffers:

```
sage: buf = io.StringIO('a
b
c')
sage: print(try_read(buf))
a
b
c  
sage: _ = buf.seek(0); try_read(buf, splitlines=True)
['a\n', 'b\n', 'c']
sage: buf = io.BytesIO(b'a
b
c')
sage: try_read(buf) == b'a
b
c'
True
sage: _ = buf.seek(0)
sage: try_read(buf, splitlines=True) == [b'a\n', b'b\n', b'c']
True
```

```
>>> from sage.all import *
>>> buf = io.StringIO('a
b
c')
>>> print(try_read(buf))
a
b
c
>>> _ = buf.seek(Integer(0)); try_read(buf, splitlines=True)
['a\n', 'b\n', 'c']
>>> buf = io.BytesIO(b'a
b
c')
>>> try_read(buf) == b'a
b
c'
True
>>> _ = buf.seek(Integer(0))
>>> try_read(buf, splitlines=True) == [b'a\n', b'b\n', b'c']
True
```

Custom readable:

```
sage: class MyFile():
    ...    def read(self): return 'Hello world!'
sage: try_read(MyFile())
'Hello world!'
sage: try_read(MyFile(), splitlines=True)
['Hello world!']
```

```
>>> from sage.all import *
>>> class MyFile():
...    def read(self): return 'Hello world!'
>>> try_read(MyFile())
'Hello world!'
>>> try_read(MyFile(), splitlines=True)
['Hello world!']
```

Not readable:
2.6.2 Miscellaneous functions (Cython)

This file contains support for products, running totals, balanced sums, and also a function to flush output from external library calls.

AUTHORS:

- William Stein (2005)
- Joel B. Mohler (2007-10-03): Reimplemented in Cython and optimized
- Robert Bradshaw (2008-03-26): Balanced product tree for generators and iterators
- Stefan van Zwam (2013-06-06): Added bitset tests, some docstring cleanup

```python
class sage.misc.misc_c.NonAssociative:
    def __init__(self, left, right=None):
        pass

    def __mul__(self, other):
        if self is other:
            return self
        if right is None:
            return NonAssociative(left=left, right=self)
        return NonAssociative(left=left, right=other)
```

EXAMPLES:

```python
sage: L = [NonAssociative(label) for label in 'abcdef']
sage: prod(L)
((a*b)*c)((d*e)*f)

sage: L = [NonAssociative(label) for label in range(20)]
sage: prod(L, recursion_cutoff=5)
((((((0*1)*2)*3)*4)*((5*6)*7)*8)*9)*((((10*11)*12)*13)*14)*((((15*16)*17)*18)*19))

sage: L = [NonAssociative(label) for label in range(14)]
sage: prod(L, recursion_cutoff=1)
((((0*1)*(2*3))*((4*5)*6))*((7*8)*(9*10))*((11*12)*13))
```
sage.misc.misc_c.balanced_sum(x, z=None, recursion_cutoff=5)

Return the sum of the elements in the list x. If optional argument z is not given, start the sum with the first element of the list, otherwise use z. The empty product is the int 0 if z is not specified, and is z if given. The sum is computed recursively, where the sum is split up if the list is greater than recursion_cutoff. recursion_cutoff must be at least 3.

This assumes that your addition is associative; we do not promise which end of the list we start at.

EXAMPLES:

```
sage: balanced_sum([1, 2, 34])
37
sage: balanced_sum([2, 3], 5)
10
sage: balanced_sum([1, 2, 3], 5)
11
```

Order should be preserved:

```
sage: balanced_sum([[i] for i in range(10)], [], recursion_cutoff=3)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We make copies when appropriate so that we do not accidentally modify the arguments:

```
sage: list(range(10^5)) == balanced_sum([[i] for i in range(10^5)], [])
True
sage: list(range(10^5)) == balanced_sum([[i] for i in range(10^5)], [])
True
```

AUTHORS:

- Joel B. Mohler (2007-10-03): Reimplemented in Cython and optimized
• Robert Bradshaw (2007-10-26): Balanced product tree, other optimizations, (lazy) generator support

\[\text{sage.misc.misc_c.cyflush()}\]

Flush any output left over from external library calls.

Starting with Python 3, some output from external libraries (like FLINT) is not flushed, and so if a doctest produces such output, the output may not appear until a later doctest. See Issue #28649.

Use this function after a doctest which produces potentially unflushed output to force it to be flushed.

EXAMPLES:

```python
sage: R.<t> = QQ[]
sage: t^(sys.maxsize//2) # does not work, because needs sage.libs.flint
Traceback (most recent call last):
    ...
RuntimeError: FLINT exception
sage: from sage.misc.misc_c import cyflush
sage: cyflush()
...
```

\[\text{sage.misc.misc_c.iterator_prod}(L, z=\text{None})\]

Attempt to do a balanced product of an arbitrary and unknown length sequence (such as a generator). Intermediate multiplications are always done with subproducts of the same size (measured by the number of original factors) up until the iterator terminates. This is optimal when and only when there are exactly a power of two number of terms.

A StopIteration is raised if the iterator is empty and \(z\) is not given.

EXAMPLES:

```python
sage: from sage.misc.misc_c import iterator_prod
sage: iterator_prod(1..5)
120
sage: iterator_prod([], z='anything')
'anything'

sage: from sage.misc.misc_c import NonAssociative
sage: L = [NonAssociative(label) for label in 'abcdef']
sage: iterator_prod(L)
((a*b)*(c*d))*(e*f))

>>> from sage.all import *
>>> from sage.misc.misc_c import iterator_prod
>>> iterator_prod(ellipsis_iter(Integer(1),Ellipsis,Integer(5)))
120
>>> iterator_prod([], z='anything')
```

(continues on next page)
'anything'

```python
>>> from sage.misc.misc_c import NonAssociative
>>> L = [NonAssociative(label) for label in 'abcdef']
>>> iterator_prod(L)
(((a*b)*(c*d))*(e*f))
```

```
sage.misc.misc_c.normalize_index(key, size)

Normalize an index key and return a valid index or list of indices within the range(0, size).

INPUT:

- key – the index key, which can be either an integer, a tuple/list of integers, or a slice.
- size – the size of the collection

OUTPUT:

A tuple (SINGLE, VALUE), where SINGLE is True (i.e., 1) if VALUE is an integer and False (i.e., 0) if VALUE is a list.

EXAMPLES:

```python
sage: from sage.misc.misc_c import normalize_index
sage: normalize_index(-6,5)
Traceback (most recent call last):
... IndexError: index out of range
sage: normalize_index(-5,5)
[0]
sage: normalize_index(-4,5)
[1]
sage: normalize_index(-3,5)
[2]
sage: normalize_index(-2,5)
[3]
sage: normalize_index(-1,5)
[4]
sage: normalize_index(0,5)
[0]
sage: normalize_index(1,5)
[1]
sage: normalize_index(2,5)
[2]
sage: normalize_index(3,5)
[3]
sage: normalize_index(4,5)
[4]
sage: normalize_index(5,5)
Traceback (most recent call last):
... IndexError: index out of range
```
```
(continues on next page)
IndexError: index out of range

```python
sage: normalize_index((-2,3),5)
[3, 3]
sage: normalize_index((5,0),5)
Traceback (most recent call last):
...
IndexError: index out of range
```

```python
sage: normalize_index((-5,2),5)
[0, 2]
sage: normalize_index((0,-2),5)
[0, 3]
sage: normalize_index((2,-3),5)
[2, 2]
sage: normalize_index((3,3),5)
[3, 3]
sage: normalize_index((-2,-5),5)
[3, 0]
sage: normalize_index((-2,-4),5)
[3, 1]
sage: normalize_index((-2,-1,3),5)
[3, 4, 3]
sage: normalize_index((4,2,1),5)
[4, 2, 1]
sage: normalize_index((-2,-3,-4),5)
[3, 2, 1]
sage: normalize_index((3,-2,-3),5)
[3, 3, 2]
sage: normalize_index((-5,2,-3),5)
[0, 2, 2]
sage: normalize_index((4,4,-5),5)
[4, 4, 0]
sage: s=slice(None, None, None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(None, None,-2); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(None, None,-2); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(None, -2, None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(None, -2, None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(None, -2, None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(None, None, None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(None, 4, None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(None, 4, None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(-2,None,None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(-2,None,-2); normalize_index(s,5)==list(range(5))[s]
True
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

```python
sage: s=slice(-2,-2,-2); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(-2,-2,4); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(-2,4,None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(-2,4,-2); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(-2,4,4); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(-2,4,4); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(4,None,None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(4,None,-2); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(4,None,4); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(4,-2,None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(4,-2,-2); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(4,-2,4); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(4,None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(4,4,None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(4,4,-2); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(4,4,4); normalize_index(s,5)==list(range(5))[s]
True
```

>>> from sage.all import *
>>> from sage.misc.misc_c import normalize_index

>>> normalize_index(-Integer(6),Integer(5))
Traceback (most recent call last):
...
IndexError: index out of range
```
```
```
>>> normalize_index(Integer(5), Integer(5))
Traceback (most recent call last):
...
IndexError: index out of range
>>> normalize_index(Integer(6), Integer(5))
Traceback (most recent call last):
...
IndexError: index out of range
>>> normalize_index((Integer(4), -Integer(6)), Integer(5))
Traceback (most recent call last):
...
IndexError: index out of range
>>> normalize_index((-Integer(2), Integer(3)), Integer(5))
[3, 3]
>>> normalize_index((Integer(5), Integer(0)), Integer(5))
Traceback (most recent call last):
...
IndexError: index out of range
>>> normalize_index((-Integer(5), Integer(2)), Integer(5))
[0, 2]
>>> normalize_index((Integer(0), -Integer(2)), Integer(5))
[0, 3]
>>> normalize_index((Integer(2), -Integer(3)), Integer(5))
[2, 2]
>>> normalize_index((Integer(3), Integer(3)), Integer(5))
[3, 3]
>>> normalize_index((-Integer(2), -Integer(5)), Integer(5))
[3, 0]
>>> normalize_index((-Integer(2), -Integer(4)), Integer(5))
[3, 1]
>>> normalize_index((-Integer(2), -Integer(1), Integer(3)), Integer(5))
[3, 4, 3]
>>> normalize_index((Integer(4), Integer(2), Integer(1)), Integer(5))
[4, 2, 1]
>>> normalize_index([-Integer(2), -Integer(3), -Integer(4)], Integer(5))
[3, 2, 1]
>>> normalize_index((Integer(3), -Integer(2), -Integer(3)), Integer(5))
[3, 3, 2]
>>> normalize_index([-Integer(5), Integer(2), -Integer(3)], Integer(5))
[0, 2, 2]
>>> normalize_index((Integer(4), Integer(4), -Integer(5)), Integer(5))
[4, 4, 0]
>>> s=slice(None, None, None); normalize_index(s, None)==list(range(Integer(5)))[s]
True
>>> s=slice(None, None, -Integer(2)); normalize_index(s, None)==list(range(Integer(5)))[s]
True
>>> s=slice(None, None, Integer(4)); normalize_index(s, None)==list(range(Integer(5)))[s]
True
>>> s=slice(None, -Integer(2), None); normalize_index(s, None)==list(range(Integer(5)))[s]
True
>>> s=slice(None, -Integer(2), -Integer(2)); normalize_index(s, None)==list(range(Integer(5)))[s]
True
(continues on next page)
>>> s=slice(\texttt{None},-\texttt{Integer(2)},\texttt{Integer(4)}); \texttt{normalize_index(s,} \rightarrow \texttt{Integer(5))=\texttt{list(range(Integer(5)))}}[\texttt{s}]
True

>>> s=slice(\texttt{None},\texttt{Integer(4)},\texttt{None}); \texttt{normalize_index(s,} \rightarrow \texttt{Integer(5))=\texttt{list(range(Integer(5)))}}[\texttt{s}]
True

>>> s=slice(\texttt{None},\texttt{Integer(4)},-\texttt{Integer(2)}); \texttt{normalize_index(s,} \rightarrow \texttt{Integer(5))=\texttt{list(range(Integer(5)))}}[\texttt{s}]
True

>>> s=slice(\texttt{None},\texttt{Integer(4)},\texttt{Integer(4)}); \texttt{normalize_index(s,} \rightarrow \texttt{Integer(5))=\texttt{list(range(Integer(5)))}}[\texttt{s}]
True

>>> s=slice(-\texttt{Integer(2)},\texttt{None},\texttt{None}); \texttt{normalize_index(s,} \rightarrow \texttt{Integer(5))=\texttt{list(range(Integer(5)))}}[\texttt{s}]
True

>>> s=slice(-\texttt{Integer(2)},-\texttt{Integer(2)},\texttt{None}); \texttt{normalize_index(s,} \rightarrow \texttt{Integer(5))=\texttt{list(range(Integer(5)))}}[\texttt{s}]
True

>>> s=slice(-\texttt{Integer(2)},-\texttt{Integer(2)},-\texttt{Integer(2)}); \texttt{normalize_index(s,} \rightarrow \texttt{Integer(5))=\texttt{list(range(Integer(5)))}}[\texttt{s}]
True

>>> s=slice(-\texttt{Integer(2)},-\texttt{Integer(2)},\texttt{Integer(4)}); \texttt{normalize_index(s,} \rightarrow \texttt{Integer(5))=\texttt{list(range(Integer(5)))}}[\texttt{s}]
True

>>> s=slice(-\texttt{Integer(2)},\texttt{Integer(4)},\texttt{None}); \texttt{normalize_index(s,} \rightarrow \texttt{Integer(5))=\texttt{list(range(Integer(5)))}}[\texttt{s}]
True

(continues on next page)
sage.misc.misc_c.prod(x, z=None, recursion_cutoff=5)

Return the product of the elements in the list x.

If optional argument z is not given, start the product with the first element of the list, otherwise use z. The empty product is the int 1 if z is not specified, and is z if given.

This assumes that your multiplication is associative; we do not promise which end of the list we start at.

See also:

For the symbolic product function, see sage.calculus.calculus.symbolic_product().

EXAMPLES:

```python
sage: prod([1, 2, 34])
68
sage: prod([2, 3], 5)
30
sage: prod((1, 2, 3), 5)
30
sage: F = factor(-2006); F
-1 * 2 * 17 * 59
sage: prod(F)
-2006
```

```python
>>> from sage.all import *
```

```python
>>> prod([Integer(1), Integer(2), Integer(34)])
68
>>> prod([Integer(2), Integer(3)], Integer(5))
30
>>> prod((Integer(1), Integer(2), Integer(3)), Integer(5))
30
>>> F = factor(-Integer(2006)); F
-1 * 2 * 17 * 59
>>> prod(F)
-2006
```

AUTHORS:

- Joel B. Mohler (2007-10-03): Reimplemented in Cython and optimized
- Robert Bradshaw (2008-03-26): Balanced product tree for generators and iterators

sage.misc.misc_c.running_total(L, start=None)

Return a list where the i-th entry is the sum of all entries up to (and including) i.

INPUT:
• \texttt{L} – the list
• \texttt{start} – (optional) a default start value

**EXAMPLES:**

```python
sage: running_total(range(5))
[0, 1, 3, 6, 10]
```

```python
sage: running_total("abcdef")
['a', 'ab', 'abc', 'abcd', 'abcde', 'abcdef']
```

```python
sage: running_total("abcdef", start="%")
['%a', '%ab', '%abc', '%abcd', '%abcde', '%abcdef']
```

```python
sage: running_total([1..10], start=100)
[101, 103, 106, 110, 115, 121, 128, 136, 145, 155]
```

```python
sage: running_total([])
[]
```

```python
>>> from sage.all import *
```

```python
>>> running_total(range(Integer(5)))
[0, 1, 3, 6, 10]
```

```python
>>> running_total("abcdef")
['a', 'ab', 'abc', 'abcd', 'abcde', 'abcdef']
```

```python
>>> running_total("abcdef", start="%")
['%a', '%ab', '%abc', '%abcd', '%abcde', '%abcdef']
```

```python
>>> running_total((ellipsis_range(Integer(1),Ellipsis,Integer(10))),...
→ start=Integer(100))
[101, 103, 106, 110, 115, 121, 128, 136, 145, 155]
```

```python
>>> running_total([])
[]
```

```python
class sage.misc.misc_c.sized_iter

Bases: object

Wrapper for an iterator to verify that it has a specified length.

INPUT:

• \texttt{iterable} – object to be iterated over
• \texttt{length} – (optional) the required length. If this is not given, then \texttt{len(iterable)} will be used.

If the iterable does not have the given length, a \texttt{ValueError} is raised during iteration.

**EXAMPLES:**

```python
sage: from sage.misc.misc_c import sized_iter
```

```python
sage: list(sized_iter(range(4)))
[0, 1, 2, 3]
```

```python
sage: list(sized_iter(range(4), 4))
[0, 1, 2, 3]
```

```python
sage: list(sized_iter(range(5), 4))
Traceback (most recent call last):
...
ValueError: sequence too long (expected length 4, got more)
```

```python
sage: list(sized_iter(range(3), 4))
Traceback (most recent call last):
...
ValueError: sequence too short (expected length 4, got 3)
```
Utilities, Release 10.4

```python
>>> from sage.all import *

>>> from sage.misc.misc_c import sized_iter

>>> list(sized_iter(range(Integer(4))))
[0, 1, 2, 3]

>>> list(sized_iter(range(Integer(4)), Integer(4)))
[0, 1, 2, 3]

>>> list(sized_iter(range(Integer(5)), Integer(4)))
Traceback (most recent call last):
...
ValueError: sequence too long (expected length 4, got more)

>>> list(sized_iter(range(Integer(3)), Integer(4)))
Traceback (most recent call last):
...
ValueError: sequence too short (expected length 4, got 3)

If the iterable is too long, we get the error on the last entry:

```sage```
 it = sized_iter(range(5), 2)
 next(it)
 0
 next(it)
 Traceback (most recent call last):
   ...
 ValueError: sequence too long (expected length 2, got more)
```

```python
>>> from sage.all import *

>>> it = sized_iter(range(Integer(5)), Integer(2))

>>> next(it)
0

>>> next(it)
Traceback (most recent call last):
...
ValueError: sequence too long (expected length 2, got more)
```

When the expected length is zero, the iterator is checked on construction:

```sage```
 list(sized_iter([], 0))
[]
 sized_iter([], 0)
 Traceback (most recent call last):
   ...
 ValueError: sequence too long (expected length 0, got more)
```

```python
>>> from sage.all import *

>>> list(sized_iter([], Integer(0)))
[]

>>> sized_iter([], Integer(0))
Traceback (most recent call last):
...
ValueError: sequence too long (expected length 0, got more)
```

If no length is given, the iterable must implement `__len__`:

```sage```
 sized_iter(x for x in range(4))
 Traceback (most recent call last):
   ...
 TypeError: object of type 'generator' has no len()
```

2.6. Miscellaneous Useful Functions 223
>>> from sage.all import *
>>> sized_iter(x for x in range(Integer(4)))
Traceback (most recent call last):
...
  TypeError: object of type 'generator' has no len()

2.6.3 Verbosity System and Logging in SageMath

Howto: Logging

Using Python’s Logging Module

Import it:

```python
sage: import logging
sage: logging.basicConfig()  # only needed once
```

Setting the level:

```python
sage: logging.getLogger().setLevel(logging.INFO)
```

Log something:

```python
sage: logger = logging.getLogger(__name__)
sage: logger.info('Hello. I am talking to you.')
INFO:__main__:Hello. I am talking to you.
```

If we haven't set the logging level to logging.INFO, then the previous wouldn't have been shown.

```python
sage: logger.debug('Hello. I am really talking a lot.')
```

The latter is not shown as the current logging level is only `logging.INFO` and not `logging.DEBUG`.

Reset the level:

```python
sage: logging.getLogger().setLevel(logging.WARNING)
```
Warnings are still shown at this default level (`logging.WARNING`):

```python
sage: logger.warning('Hello. I am warning you.')
WARNING:__main__:Hello. I am warning you.
```

And that's all.

There are a lot more features, see Logging facility for Python.

**Using SageMath’s Verbosity System**

Alternatively, this module provides `verbose()`, `set_verbose()`, `get_verbose()` which can be used as follows:

```python
sage: from sage.misc.verbose import verbose, set_verbose, get_verbose

sage: set_verbose(1)

sage: t = verbose("This is SageMath.", level=0)
verbose 0 (<module>) This is SageMath.

sage: t = verbose("This is SageMath.", level=1)
verbose 1 (<module>) This is SageMath.

sage: t = verbose("This is SageMath.", level=2)
```

**Logging Levels of SageMath and Python**

<table>
<thead>
<tr>
<th>SageMath</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>−2</td>
<td><code>logging.CRITICAL</code></td>
</tr>
<tr>
<td>−1</td>
<td><code>logging.ERROR</code></td>
</tr>
<tr>
<td>0</td>
<td><code>logging.WARNING</code></td>
</tr>
<tr>
<td>1</td>
<td><code>logging.INFO</code></td>
</tr>
<tr>
<td>2</td>
<td><code>logging.DEBUG</code></td>
</tr>
</tbody>
</table>
Various

AUTHORS:

  • Daniel Krenn (2016)

Functions

sage.misc.verbose.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:

```python
sage: get_verbose()
0
sage: set_verbose(2)
sage: get_verbose()
2
sage: set_verbose(0)
```

sage.misc.verbose.get_verbose_files()

sage.misc.verbose.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:

```python
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)
```
Utilities, Release 10.4

```python
>>> from sage.all import *
>>> set_verbose(Integer(2))
>>> verbose("This is Sage.", level=Integer(1))  # not tested
VERBOISE1 (?): This is Sage.
>>> verbose("This is Sage.", level=Integer(2))  # not tested
VERBOISE2 (?): This is Sage.
>>> verbose("This is Sage.", level=Integer(3))  # not tested
[no output]
>>> set_verbose(Integer(0))
```

```python
sage.misc.verbose.set_verbose_files(file_name)
sage.misc.verbose.unset_verbose_files(file_name)
sage.misc.verbose.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()

EXAMPLES:

```python
sage: set_verbose(1)
sage: t = cputime()
sage: t = verbose("This is Sage.", t, level=1, caller_name="william")  # not tested
VERBOISE1 (william): This is Sage. (time = 0.0)
sage: set_verbose(0)
```

```python
>>> from sage.all import *
>>> set_verbose(Integer(1))
>>> t = cputime()
>>> t = verbose("This is Sage.", t, level=Integer(1), caller_name="william")  # not tested
VERBOISE1 (william): This is Sage. (time = 0.0)
>>> set_verbose(Integer(0))
```

### 2.7 Laziness

#### 2.7.1 Lazy attributes

AUTHORS:
- Nicolas Thiery (2008): Initial version
- Nils Bruin (2013-05): Cython version
class sage.misc.lazy_attribute.lazy_attribute(f)

Bases: _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`:

```python
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 in A")
    ....:         return self.a + 1
    ....:
```

```python
>>> from sage.all import *
>>> class A():
...     def __init__(self):
...         self.a=Integer(2) # just to have some data to calculate from
...     ...
...     @lazy_attribute
...     def x(self):
...         print("calculating x in A")
...         return self.a + Integer(1)
...     ...
```

For an instance `a` of `A`, `a.x` is calculated the first time it is accessed, and then stored as a usual attribute:

```python
sage: a = A()
sage: a.x
calculating x in A
3
sage: a.x
3
```

```python
>>> from sage.all import *
>>> a = A()
>>> a.x
calculating x in A
3
>>> 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:

```python
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
```

```python
>>> from sage.all import *
>>> a = A()
>>> a.__dict__
{'a': 2}
>>> a.x
calculating x in A
3
>>> a.__dict__
{'a': 2, 'x': 3}
>>> a.x
3
>>> 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:

```python
sage: a = A()
sage: a.x = 4
sage: a.x
4
sage: a.__dict__
{'a': 2, 'x': 4}
```

```python
>>> from sage.all import *
>>> a = A()
>>> a.x = Integer(4)
>>> a.x
4
>>> a.__dict__
{'a': 2, 'x': 4}
```

Class binding results in the lazy attribute itself:

```python
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:

```python
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
```

```python
>>> from sage.all import *
>>> 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

>>> b = B()
>>> b.x
y not there; B does not define x
calculating x in A
3
>>> b = B()
>>> b.y = Integer(1)
>>> 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:

```python
sage: a = A()
```
```
sage: hasattr(a, "x")
calculating x in A
True
```

```python
>>> from sage.all import *
>>> a = A()
>>> 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:

```python
sage: class A:
    ...
    @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
    ...
```

```python
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
```

```python
>>> from sage.all import *
>>> class A:
...     @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 Integer(3)
...>>> a = A()
>>> hasattr(a, "x") # todo: not implemented
testing for x existence
>>> a.x
calculating x in A
3
>>> a.x
3
```
Here is a full featured example, with both conditional definition and existence testing:

```python
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: b.x
# todo: not implemented
y not there; B does not define x
testing for x existence
True

sage: b = B()
sage: b.y = 1
sage: b.x
# todo: not implemented
testing for x existence in B
True

sage: b = B()
sage: b.x
y not there; B does not define x
calculating x in A
3

>>> from sage.all import *

>>> 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
...:

>>> b = B()

>>> b.x
# todo: not implemented
y not there; B does not define x
testing for x existence
True

>>> b.x
y not there; B does not define x
calculating x from y in B
1

(continues on next page)
lazy attributes and introspection

Todo: Make the following work nicely:

```
>>> from sage.all import *
>>> b.x             # todo: not implemented
5

sage: A().len
5
```

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.

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 Issue #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):",

```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

```python
....: " Element = MyElement"]
sage: cython('
'.join(cython_code))  #...
→ needs sage.misc.cython
sage: P = MyParent(category=Rings())  #...
→ needs sage.misc.cython
sage: P.element_class  # indirect doctest  #...
→ needs sage.misc.cython
<class '...MyElement'>
```

```python
>>> from sage.all import *
>>> 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"]
>>> cython('
'.join(cython_code))  #...
→ needs sage.misc.cython
>>> P = MyParent(category=Rings())  #...
→ needs sage.misc.cython
>>> P.element_class  # indirect doctest  #...
→ needs sage.misc.cython
<class '...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:

```python
sage: class descriptor():
....: def __get__(self, obj, cls):
....:     print(cls)
....:     return 1
sage: class A():
....:     x = descriptor()
sage: class B(A):
....:     pass
....:
```

```python
>>> from sage.all import *
>>> class descriptor():
...     def __get__(self, obj, cls):
...         print(cls)
...         return Integer(1)
>>> class A():
...     x = descriptor()
>>> class B(A):
...     pass
....:
```

This is fine:

```python
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():
 sleepy: @lazy_attribute
 sleepy: def unimplemented_A(self):
 sleepy: return NotImplemented
 sleepy: @lazy_attribute
 sleepy: def unimplemented_AB(self):
 sleepy: return NotImplemented
 sleepy: @lazy_attribute
 sleepy: def unimplemented_BImplemented_A(self):
 sleepy: return 1

sage: class B(A):
 sleepy: @lazy_attribute
 sleepy: def unimplemented_B(self):
 sleepy: return NotImplemented
 sleepy: @lazy_attribute
 sleepy: def unimplemented_AB(self):
 sleepy: return NotImplemented
 sleepy: @lazy_attribute
 sleepy: def unimplemented_BImplemented_A(self):
 sleepy: return NotImplemented

sage: class C(B):
 sleepy: pass
```
Utilities, Release 10.4

>>> from sage.all import *

>>> class A():
...     @lazy_attribute
...     def unimplemented_A(self):
...         return NotImplemented

>>> class B(A):
...     @lazy_attribute
...     def unimplemented_B(self):
...         return NotImplemented

>>> 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  # todo: not implemented
Traceback (most recent call last):
  ...
```

(continues on next page)
AttributeError: '...' object has no attribute 'unimplemented_A'

```python
sage: B().unimplemented_A
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_B'
```

```python
sage: B().unimplemented_AB
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_AB'
```

```python
sage: B().unimplemented_B_implemented_A
1
```

```python
sage: C().unimplemented_A()
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_A'
```

```python
sage: C().unimplemented_B()
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_B'
```

```python
sage: C().unimplemented_AB()
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_AB'
```

```python
sage: C().unimplemented_B_implemented_A
# todo: not implemented
1
```

```python
>>> from sage.all import *
>>> B().unimplemented_A
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_A'
```

```python
>>> B().unimplemented_B
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_B'
```

```python
>>> B().unimplemented_AB
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_AB'
```

```python
>>> B().unimplemented_B_implemented_A
1
```

```python
>>> C().unimplemented_A()
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_A'
```

```python
>>> C().unimplemented_B()
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_B'
```

```python
>>> C().unimplemented_AB()
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_AB'
```

```python
>>> C().unimplemented_B_implemented_A
# todo: not implemented
1
```
class sage.misc.lazy_attribute.lazy_class_attribute(f)
Bases: lazy_attribute

A lazy class attribute for a 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 through one of its objects.

This is very similar to \texttt{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:

```python
sage: class Cl():
    ...  @lazy_class_attribute
    ...  def x(cls):
    ...      print("computing x")
    ...      return 1
sage: Cl.x
computing x
1
sage: Cl.x
1
```

As for a usual class attribute it is also possible to access it from an object:

```python
sage: b = Cl()
sage: b.x
1
```

```
>>> from sage.all import *
>>> class Cl():
...    ...  @lazy_class_attribute
...    ...  def x(cls):
...    ...      print("computing x")
...    ...      return Integer(1)
>>> Cl.x
computing x
1
```

First access from an object also properly triggers the computation:

```python
sage: class C11():
    ...  @lazy_class_attribute
    ...  def x(cls):
    ...      print("computing x")
    ...      return 1
sage: C11().x
computing x
1
sage: C11().x
1
```
>>> from sage.all import *

>>> class C11():
...     @lazy_class_attribute
...     def x(cls):
...         print("computing x")
...         return Integer(1)

>>>

Cl1().x
computing x
1

Cl1().x
1

Warning: The behavior of lazy class attributes with respect to inheritance is not specified. It currently depends on the evaluation order:

```python
class A():
     @lazy_class_attribute
     def x(cls):
         print("computing x")
         return str(cls)

     @lazy_class_attribute
     def y(cls):
         print("computing y")
         return str(cls)

class B(A):
     pass
```

```python
A.x
computing x
"<class '__main__.A'>"

B.x
"<class '__main__.A'>"

B.y
computing y
"<class '__main__.B'>"

A.y
computing y
"<class '__main__.A'>"

B.y
"<class '__main__.B'>"
```
2.7.2 Lazy format strings

class sage.misc.lazy_format.LazyFormat

    Bases: str

    Lazy format strings

Note: We recommend to use sage.misc.lazy_string.lazy_string() instead, which is both faster and more flexible.

An instance of LazyFormat behaves like a usual format string, except that the evaluation of the __repr__ method of the formatted arguments it postponed until actual printing.

EXAMPLES:

Under normal circumstances, Lazyformat strings behave as usual:

```python
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:

```python
sage: class IDontLikeBeingPrinted():
    ...:     def __repr__(self):
    ...:         raise ValueError("do not ever try to print me")
```

There is no error when binding a lazy format with the broken object:

```python
sage: lf = LazyFormat("<%s>")%IDontLikeBeingPrinted()
```

The error only occurs upon printing:
Common use case:

Most of the time, \texttt{\_\_repr\_} methods are only called during user interaction, and therefore need not be fast; and indeed there are objects \(x\) in Sage such \(x.\texttt{\_\_repr\_}()\) 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 \texttt{unittest} or \texttt{TestSuite} executions:

```python
sage: QQ._tester().assertIn(0, QQ,
....:    "\$s doesn't contain 0"%QQ)
```

```python
>>> from sage.all import *
>>> QQ._tester().assertIn(Integer(0), QQ,
...    "\$s doesn't contain 0"%QQ)
```

In the above \(QQ.\texttt{\_\_repr\_}()\) has been called, and the result immediately discarded. To demonstrate this we replace \(QQ\) in the format string argument with our broken object:

```python
sage: QQ._tester().assertTrue(True,
....:    "\$s doesn't contain 0"%IDontLikeBeingPrinted())
Traceback (most recent call last):
  ...:  ValueError: do not ever try to print me
```

```python
>>> from sage.all import *
>>> QQ._tester().assertTrue(True,
...    "\$s doesn't contain 0"%IDontLikeBeingPrinted())
Traceback (most recent call last):
  ...:  ValueError: do not ever try to print me
```

This behavior can induce major performance penalties when testing. Note that this issue does not impact the usual assert:

```python
sage: assert True, "\$s is wrong"%IDontLikeBeingPrinted()
```

```python
>>> from sage.all import *
>>> assert True, "\$s is wrong"%IDontLikeBeingPrinted()
```

We now check that \texttt{LazyFormat} indeed solves the assertion problem:

```python
sage: QQ._tester().assertTrue(True,
....:    LazyFormat("\$s is wrong")%IDontLikeBeingPrinted())
sage: QQ._tester().assertTrue(False,
....:    LazyFormat("\$s is wrong")%IDontLikeBeingPrinted())
```

(continues on next page)
2.7.3 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:

```python
sage: lazy_import('sage.rings.integer_ring', 'ZZ')
sage: type(ZZ)
<class '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:

```python
sage: lazy_import('sage.rings.integer_ring', '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:

```python
sage: ZZ
doctest:warning...
UserWarning: Option `at_startup=True` for lazy import ZZ not needed anymore
```

Integer Ring
See also:

`lazy_import()`, `LazyImport`  

AUTHOR:  
• Robert Bradshaw

```python  
>>> from sage.all import  
>>> ZZ  
doctest:warning...  
UserWarning: Option `at_startup=True` for lazy import ZZ not needed anymore  
Integer Ring

 robber Bradshaw

class sage.misc.lazy_import.LazyImport  
Bases: object  

EXAMPLES:

```python  
```
sage.misc.lazy_import.clean_namespace(namespace=None)

Adjust LazyImport bindings in given namespace to refer to this actual namespace.

When LazyImport objects are imported into other namespaces via normal import instructions, the data stored on a LazyImport object that helps it to adjust the binding in the namespace to the actual imported object upon access is not adjusted. This routine fixes that.

INPUT:

* namespace – the namespace where importing the names; by default, import the names to current namespace

EXAMPLES:

sage: # needs sage.symbolic
sage: from sage.misc.lazy_import import attributes, clean_namespace
sage: from sage.calculus.calculus import maxima as C
sage: attributes(C)['_as_name']
'maxima'
sage: attributes(C)['_namespace'] is sage.calculus.calculus.__dict__
True
sage: clean_namespace(globals())
sage: attributes(C)['_as_name']
'C'
sage: attributes(C)['_namespace'] is globals()
True

(continues on next page)
sage.misc.lazy_import.ensure_startup_finished()
Make sure that the startup phase is finished.
In contrast to finish_startup(), this function can be called repeatedly.

sage.misc.lazy_import.finish_startup()
Finish the startup phase.
This function must be called exactly once at the end of the Sage import process (all).

sage.misc.lazy_import.get_star_imports(module_name)
Look up the list of names in a module that would be imported with “import *” either via a cache or actually importing.
EXAMPLES:

```python
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')  # needs sage.schemes
True
```

```python
>>> from sage.all import *
>>> from sage.misc.lazy_import import get_star_imports
>>> 'get_star_imports' in get_star_imports('sage.misc.lazy_import')
True
>>> 'EllipticCurve' in get_star_imports('sage.schemes.all')  # needs sage.schemes
True
```

sage.misc.lazy_import.is_during_startup()
Return whether Sage is currently starting up.

OUTPUT:
Boolean

sage.misc.lazy_import.lazy_import(module, names, as_=None, at_startup=False, namespace=None, deprecation=None, feature=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 names of the objects in the importing module.
  This is analogous to from ... import ... as ....
- at_startup – a boolean (default: False); whether the lazy import is supposed to be resolved at startup time
- namespace – the namespace where importing the names; by default, import the names to current namespace

2.7. Laziness
- **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 (issue_number, message)

- **feature** – a python module (optional), if it cannot be imported an appropriate error is raised

See also:

`sage.misc.lazy_import`, `LazyImport`

**EXAMPLES:**

```python
sage: lazy_import('sage.rings.integer_ring', 'ZZ')
sage: type(ZZ)
<class 'sage.misc.lazy_import.LazyImport'>
sage: ZZ(4.0) 4
sage: lazy_import('sage.rings.real_double', 'RDF', 'my_RDF')
sage: my_RDF._get_object() is RDF
True
sage: my_RDF(1/2)
0.5
sage: lazy_import('sage.rings.rational_field', ['QQ', 'frac'], ['my_QQ', 'my_frac'])
sage: my_QQ._get_object() is QQ
True
sage: my_frac._get_object() is sage.rings.rational_field.frac
True
```

```python
>>> from sage.all import *
>>> lazy_import('sage.rings.integer_ring', 'ZZ')
>>> type(ZZ)
<class 'sage.misc.lazy_import.LazyImport'>
>>> ZZ(RealNumber(4.0))
4
>>> lazy_import('sage.rings.real_double', 'RDF', 'my_RDF')
>>> my_RDF._get_object() is RDF
True
>>> my_RDF(Integer(1)/Integer(2))
0.5
>>> lazy_import('sage.rings.rational_field', ['QQ', 'frac'], ['my_QQ', 'my_frac'])
>>> my_QQ._get_object() is QQ
True
>>> my_frac._get_object() is sage.rings.rational_field.frac
True
```

Upon the first use, the object is injected directly into the calling namespace:

```python
sage: lazy_import('sage.rings.integer_ring', 'ZZ', 'my_ZZ')
sage: my_ZZ
sage: my_ZZ(37)
37
sage: my_ZZ is ZZ
False
```
We check that `lazy_import()` also works for methods:

```python
sage: class Foo:
    ....:     lazy_import('sage.plot.plot', 'plot')
sage: class Bar(Foo):
    ....:         pass
sage: type(Foo.__dict__['plot'])
<... function>
```

If deprecated then a deprecation warning is issued:

```python
sage: lazy_import('sage.rings.padics.factory', 'Qp', 'my_Qp',
    ....:     deprecation=Integer(14275))
```

(continues on next page)
please use "from sage.rings.padics.factory import Qp as my_Qp" instead. See https://github.com/sagemath/sage/issues/14275 for details.

5-adic Field with capped relative precision 20

An example of deprecation with a message:

```python
sage: lazy_import('sage.rings.padics.factory', 'Qp', 'my_Qp_msg',
            deprecation=(14275, "This is an example."))
sage: my_Qp_msg(5)
# needs sage.rings.padics
doctest:...: DeprecationWarning: This is an example.
See https://github.com/sagemath/sage/issues/14275 for details.
5-adic Field with capped relative precision 20
```

An example of an import relying on a feature:

```python
sage: from sage.features import PythonModule
sage: lazy_import('ppl', 'equation',
            feature=PythonModule('ppl', spkg='pplpy', type='standard'))
sage: equation
# needs pplpy
<cyfunction equation at ...>
sage: lazy_import('PyNormaliz', 'NmzListConeProperties',
            feature=PythonModule('PyNormaliz', spkg='pynormaliz'))
sage: NmzListConeProperties
<built-in function NmzListConeProperties>
sage: lazy_import('foo', 'not_there',
            feature=PythonModule('foo', spkg='non-existing-package'))
sage: not_there
Failed lazy import:
foo is not available.
Importing not_there failed: No module named 'foo'...
No equivalent system packages for ... are known to Sage...
```

```python
>>> from sage.all import *
>>> lazy_import('sage.rings.padics.factory', 'Qp', 'my_Qp_msg',
            deprecation=(Integer(14275), "This is an example."))
>>> my_Qp_msg(Integer(5))
# needs sage.rings.padics
doctest:...: DeprecationWarning: This is an example.
See https://github.com/sagemath/sage/issues/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.

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: lazy_import('sage.rings.integer_ring', 'ZZ', 'my ZZ')
    sage: my ZZ(Integer(123))
    doctest:warning...
    UserWarning: Resolving lazy import ZZ during startup
    123
    sage: sage.misc.lazy_import.finish_startup()

    >>> from sage.all import *
    >>> sage.misc.lazy_import.test_fake_startup()
    >>> lazy_import('sage.rings.integer_ring', 'ZZ', 'my ZZ')
    >>> my ZZ(Integer(123))
    doctest:warning...
    UserWarning: Resolving lazy import ZZ during startup
    123
    >>> sage.misc.lazy_import.finish_startup()

2.7.4 Lazy import cache

sage.misc.lazy_import_cache.get_cache_file()
    Return the canonical filename for caching names of lazily imported modules.

    EXAMPLES:

    sage: from sage.misc.lazy_import_cache import get_cache_file
    sage: get_cache_file() ('...-lazy_import_cache.pickle')
    sage: get_cache_file().startswith(DOT_SAGE)
    True
    sage: 'cache' in get_cache_file()
    True

    >>> from sage.all import *
    >>> from sage.misc.lazy_import_cache import get_cache_file
    >>> get_cache_file() ('...-lazy_import_cache.pickle')
    >>> get_cache_file().startswith(DOT_SAGE)
    True

    (continues on next page)
It should not matter whether DOT_SAGE ends with a slash:

```
sage: OLD = DOT_SAGE
sage: sage.misc.lazy_import_cache.DOT_SAGE = '/tmp'
sage: get_cache_file().startswith('/tmp/')
True
sage: sage.misc.lazy_import_cache.DOT_SAGE = OLD
```

```
>>> from sage.all import *
```

```
>>> OLD = DOT_SAGE
```

```
>>> sage.misc.lazy_import_cache.DOT_SAGE = '/tmp'
```

```
>>> get_cache_file().startswith('/tmp/')
True
```

```
>>> sage.misc.lazy_import_cache.DOT_SAGE = OLD
```

## 2.7.5 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]  # needs sage.libs.pari
547
sage: P[10:34]  # needs sage.libs.pari
lazy list [31, 37, 41, ...]
sage: P[12:23].list()  # needs sage.libs.pari
[41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83]
sage: f = lazy_list((i**2 - 3*i for i in range(10)))
sage: print(" ".join(str(i) for i in f))
0 -2 -2 0 4 10 18 28 40 54
sage: i1 = iter(f)
sage: i2 = iter(f)
sage: [next(i1), next(i1)]  # needs sage.libs.pari
[0, -2]
sage: [next(i2), next(i2)]  # needs sage.libs.pari
[-2, -2]
```

```
>>> from sage.all import *
>>> from sage.misc.lazy_list import lazy_list
```

```
>>> P = lazy_list(Primes())
```

```
>>> P[Integer(100)] (continues on next page)
```
# needs sage.libs.pari

>>> P[Integer(10):Integer(34)]
lazy list [31, 37, 41, ...]

>>> P[Integer(12):Integer(23)].list()
[41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83]

>>> f = lazy_list((i**Integer(2) - Integer(3)*i for i in range(Integer(10))))
>>> print(" ".join(str(i) for i in f))
0 -2 -2 0 4 10 18 28 40 54

>>> i1 = iter(f)
>>> i2 = iter(f)
>>> [next(i1), next(i1)]
[0, -2]

>>> [next(i2), next(i2)]
[0, -2]

>>> [next(i1), next(i2)]
[-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
```

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

You can easily create your own class inheriting from `lazy_list_generic`. You should call the `lazy_list_generic` constructor (optionally with some precomputed values for the cache) and implement the method `_new_slice` that returns a new chunk of data at each call. Here is an example of implementation of the Thue–Morse word that is obtained as the fixed point of the substitution $0 \rightarrow 01$ and $1 \rightarrow 10$:

```
sage: from sage.misc.lazy_list import lazy_list_generic
sage: class MyThueMorseWord(lazy_list_generic):
    ....:    def __init__(self):
    ....:        self.i = 1

sage: from sage.misc.lazy_list import lazy_list_generic
sage: class MyThueMorseWord(lazy_list_generic):
   ....:       def __init__(self):
   ....:           self.i = 1
```

(continues on next page)
....:     lazy_list_generic.__init__(self, cache=[0,1])
....:     def __new_slice__(self):
....:         letter = self.get(self.i)
....:         self.i += 1
....:         return [0,1] if letter == 0 else [1,0]
sage: w = MyThueMorseWord()
sage: w
lazy list [0, 1, 1, ...]
sage: all(w[i] == ZZ(i).popcount()%2 for i in range(100))
True
sage: w[:500].list() == w[:1000:2].list()
True

Alternatively, you can create the lazy list from an update function:

sage: def thue_morse_update(values):
    n = len(values)
    if n == 0:
        letter = 0
    else:
        assert n%2 == 0
        letter = values[n//2]
        values.append(letter)
        values.append(1-letter)
sage: w2 = lazy_list(update_function=thue_morse_update)
sage: w2
lazy list [0, 1, 1, ...]
sage: w2[:500].list() == w[:500].list()
True

>>> from sage.all import *
>>> def thue_morse_update(values):
...     n = len(values)
...     if n == Integer(0):
...         letter = Integer(0)
...     else:
...         assert n%Integer(2) == Integer(0)
...letter = values[n//Integer(2)]
...values.append(letter)
...values.append(Integer(1)-letter)

>>> w2 = lazy_list(update_function=thue_morse_update)

>>> w2
lazy list [0, 1, 1, ...]

>>> w2[:Integer(500)].list() == w[:Integer(500)].list()
True

You can also create user-defined classes (Python) and extension types (Cython) inheriting from `lazy_list_generic`. In that case you would better implement directly the method `_update_cache_up_to`. See the examples in this file with the classes `lazy_list_from_iterator` and `lazy_list_from_function`.

### Classes and Methods

`sage.misc.lazy_list.lazy_list(data=None, initial_values=None, start=None, stop=None, step=None, update_function=None)`

Return a lazy list.

**INPUT:**

- `data` – data to create a lazy list from. This can be
  1. a (possibly infinite) iterable,
  2. a function (that takes as input an integer \( n \) and return the \( n \)-th term of the list),
  3. or a standard Python container list or tuple.
- `initial_values` – the beginning of the sequence that will not be computed from the `data` provided.
- `update_function` – you can also construct a lazy list from a function that takes as input a list of precomputed values and updates it with some more values.

**Note:** If you want finer tuning of the constructor you can directly instantiate the classes associated to lazy lists that are `lazy_list_generic`, `lazy_list_from_iterator`, `lazy_list_from_function`.

**EXAMPLES:**

The basic construction of lazy lists.

```python
sage: from sage.misc.lazy_list import lazy_list

>>> from sage.all import *
>>> from sage.misc.lazy_list import lazy_list
```

1. **Iterators:**

```python
sage: from itertools import count
sage: lazy_list(count())
lazy list [0, 1, 2, ...]
```

```python
>>> from sage.all import *
>>> from itertools import count
```
2. Functions:

```python
sage: lazy_list(lambda n: (n**2) % 17)
lazy list [0, 1, 4, ...]
```

```python
>>> from sage.all import *
```

```python
>>> lazy_list(lambda n: (n**Integer(2)) % Integer(17))
lazy list [0, 1, 4, ...]
```

3. Plain lists:

```python
sage: lazy_list([1,5,7,2])
lazy list [1, 5, 7, ...]
```

```python
>>> from sage.all import *
```

```python
>>> lazy_list([Integer(1),Integer(5),Integer(7),Integer(2)])
lazy list [1, 5, 7, ...]
```

If a function is only defined for large values, you can provide the beginning of the sequence manually:

```python
sage: l = lazy_list(divisors, [None])
sage: l
lazy list [None, [1], [1, 2], ...
```

```python
>>> from sage.all import *
```

```python
>>> l = lazy_list(divisors, [None])
```

```python
>>> l
lazy list [None, [1], [1, 2], ...
```

Lazy lists behave like lists except that they are immutable:

```python
sage: l[3::5]
lazy list [None, [1], [1, 2, 4, 8], [1, 13], ...
```

```python
>>> from sage.all import *
```

```python
>>> l=Integer(30):Integer(5).list()
lazy list [None, [1], [1, 2, 4, 8], [1, 13], ...
```

If your lazy list is finite, you can obtain the underlying list with the method `list()`:

```python
sage: l[30:50:5].list()
[[1, 2, 3, 5, 6, 10, 15, 30],
 [1, 5, 7, 35],
 [1, 2, 4, 5, 8, 10, 20, 40],
 [1, 3, 5, 9, 15, 45]]
```

```python
>>> from sage.all import *
```

```python
>>> l=Integer(30):Integer(50):Integer(5).list()
[[1, 2, 3, 5, 6, 10, 15, 30],
 [1, 5, 7, 35],
 [1, 2, 4, 5, 8, 10, 20, 40],
 [1, 3, 5, 9, 15, 45]]
```
Return a string representation of \( L \).

**INPUT:**

- \( L \) – an iterable object
- `name` – (default: 'lazy list') a string appearing at first position (i.e., in front of the actual values) in the representation
- `opening_delimiter` – (default: '[') a string heading the shown entries
- `closing_delimiter` – (default: ']') a string trailing the shown entries
- `separator` – (default: ', ') a string appearing between two entries
- `more` – (default: '...') a string indicating that not all entries of the list are shown
- `preview` – (default: 3) an integer specifying the number of elements shown in the representation string

**OUTPUT:**

A string.

**EXAMPLES:**

```python
def lazy_list_formatter(L, name='lazy list', separator=', ', more='...',
    opening_delimiter='[', closing_delimiter=']', preview=3):
    Return a string representation of L.

INPUT:

- \( L \) – an iterable object
- `name` – (default: 'lazy list') a string appearing at first position (i.e., in front of the actual values) in the representation
- `opening_delimiter` – (default: '[') a string heading the shown entries
- `closing_delimiter` – (default: ']') a string trailing the shown entries
- `separator` – (default: ', ') a string appearing between two entries
- `more` – (default: '...') a string indicating that not all entries of the list are shown
- `preview` – (default: 3) an integer specifying the number of elements shown in the representation string

OUTPUT:

A string.

EXAMPLES:

```python
def lazy_list_formatter(L, name='lazy list', separator=', ', more='...',
    opening_delimiter='[', closing_delimiter=']', preview=3):
```
```python
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> from sage.misc.lazy_list import lazy_list
>>> L = lazy_list(Primes()); L
lazy list [2, 3, 5, ...]
>>> repr(L) == lazy_list_formatter(L)
True
>>> lazy_list_formatter(L, name='primes')
'primes [2, 3, 5, ...]'
>>> lazy_list_formatter(L, opening_delimiter='(', closing_delimiter=')')
'lazy list (2, 3, 5, ...)'
>>> lazy_list_formatter(L, opening_delimiter=' ', closing_delimiter='')
'lazy list 2, 3, 5, ...'
>>> lazy_list_formatter(L, separator='--')
'lazy list [2--3--5--...]'
>>> lazy_list_formatter(L, more='and more')
'lazy list [2, 3, 5, and more]'
>>> lazy_list_formatter(L, preview=Integer(10))
'lazy list [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ...]'
>>> lazy_list_formatter(L, name='primes',
... opening_delimiter=' ',
... closing_delimiter=' ',
... separator=' ',
... more='->',
... preview=Integer(7))
'primes 2 3 5 7 11 13 17 ->'
```

class sage.misc.lazy_list.lazy_list_from_function

Bases: lazy_list_generic

INPUT:

- function – a function that maps \( n \) to the element at position \( n \). (This function only needs to be defined for length larger than the length of the cache.)
- cache – an optional list to be used as the cache. Be careful that there is no copy.
- stop – an optional integer to specify the length of this lazy list. (Otherwise it is considered infinite).

EXAMPLES:

```python
sage: from sage.misc.lazy_list import lazy_list_from_function

sage: lazy_list_from_function(euler_phi)
# needs sage.libs.pari
lazy list [0, 1, 1, ...]
sage: lazy_list_from_function(divisors, [None])
lazy list [None, [1], [1, 2], ...]
```

class sage.misc.lazy_list.lazy_list_from_iterator

Bases: lazy_list_generic

Lazylist built from an iterator.

EXAMPLES:

```python
>>> from sage.all import *
>>> from sage.misc.lazy_list import lazy_list_from_iterator
>>> lazy_list_from_iterator(euler_phi)
# needs...
lazy list [0, 1, 1, ...]
>>> lazy_list_from_iterator(divisors, [None])
lazy list [None, [1], [1, 2], ...]
```

256 Chapter 2. Programming Utilities
```python
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())[8:20551:2]
sage: m2
lazy list [8, 10, 12, ...]

sage: x = iter(m)
sage: [next(x), next(x), next(x)]
[0, 1, 2]

sage: y = iter(m)
sage: [next(y), next(y), next(y)]
[0, 1, 2]

sage: [next(x), next(y)]
[3, 3]

sage: loads(dumps(m))
lazy list [0, 1, 2, ...]
```

```python
>>> from sage.all import *
>>> from sage.misc.lazy_list import lazy_list
>>> from itertools import count

>>> m = lazy_list(count()); m
lazy list [0, 1, 2, ...]

>>> m2 = lazy_list(count())[Integer(8):Integer(20551):Integer(2)]

>>> m2
lazy list [8, 10, 12, ...]

>>> x = iter(m)

>>> [next(x), next(x), next(x)]
[0, 1, 2]

>>> y = iter(m)

>>> [next(y), next(y), next(y)]
[0, 1, 2]

>>> [next(x), next(y)]
[3, 3]

>>> loads(dumps(m))
lazy list [0, 1, 2, ...]
```

class sage.misc.lazy_list.lazy_list_from_update_function

Bases: lazy_list_generic

INPUT:

- function -- a function that updates a list of precomputed values. The update function should take as input a list and make it longer (using either the methods `append` or `extend`). If after a call to the update function the list of values is shorter a `RuntimeError` will occur. If no value is added then the lazy list is considered finite.

- cache -- an optional list to be used as the cache. Be careful that there is no copy.

- stop -- an optional integer to specify the length of this lazy list (otherwise it is considered infinite)

class sage.misc.lazy_list.lazy_list_generic

Bases: object

A lazy list
 EXAMPLES:

```
sage: from sage.misc.lazy_list import lazy_list
sage: l = lazy_list(Primes())
sage: l
# needs sage.libs.pari
˓→ lazy list [2, 3, 5, ...]
sage: l[200]
# needs sage.libs.pari
˓→ 1229
```

```
>>> from sage.all import *
>>> from sage.misc.lazy_list import lazy_list
>>> l = lazy_list(chain(iter([1,2,3]), repeat(a)))
>>> l.get(0)
1
>>> l.get(3)
'a'
>>> l.get(0)
1
>>> l.get(4)
'a'
```

```
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: unable to convert rational 1/2 to an integer
```

```
>>> from sage.all import *
>>> from sage.misc.lazy_list import lazy_list
>>> from itertools import chain, repeat
>>> f = lazy_list(chain(iter([Integer(1),Integer(2),Integer(3)]), repeat('a')))
```

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.
list()

Return the list made of the elements of self.

Note: If the iterator is sufficiently large, this will build a list of length PY_SSIZE_T_MAX which should be beyond the capacity of your RAM!

EXAMPLES:

```
sage: from sage.misc.lazy_list import lazy_list
sage: P = lazy_list(Primes())
sage: P[2:143:5].list()    # needs sage.libs.pari
```

```
sage: P = lazy_list(iter([Integer(1),Integer(2),Integer(3)]))
sage: P.list()  
[1, 2, 3]
```

```
>>> from sage.all import *
>>> from sage.misc.lazy_list import lazy_list
>>> P = lazy_list(Primes())
>>> P[Integer(2):Integer(143):Integer(5)].list()  # needs sage.libs.pari
```

```
>>> P = lazy_list(iter([Integer(1),Integer(2),Integer(3)]))
>>> P.list()  
[1, 2, 3]
```
sage.misc.lazy_list.slice_unpickle\(master, start, stop, step\)

Unpickle helper

### 2.7.6 Lazy strings

Based on speaklater: [https://github.com/mitsuhiko/speaklater](https://github.com/mitsuhiko/speaklater).

A lazy string is an object that behaves almost exactly like a string but where the value is not computed until needed. To define a lazy string you specify a function that produces a string together with the appropriate arguments for that function.

**EXAMPLES:**

```python
sage: from sage.misc.lazy_string import lazy_string
sage: L = []

sage: s = lazy_string(lambda x: str(len(x)), L)

sage: L.append(5)

sage: s
1'1'
```

Note that the function is recomputed each time:

```python
sage: L.append(6)

sage: s
l'2'
```

```python
sage: L.append(Integer(6))

sage: s
l'2'
```

sage.misc.lazy_string.is_lazy_string\(obj\)

Checks if the given object is a lazy string.

**EXAMPLES:**

```python
sage: from sage.misc.lazy_string import lazy_string, is_lazy_string
sage: f = lambda: "laziness"

sage: s = lazy_string(f)

sage: is_lazy_string(s)
True
```
```python
>>> from sage.all import *
>>> from sage.misc.lazy_string import lazy_string, is_lazy_string

f = lambda: "laziness"
s = lazy_string(f)
is_lazy_string(s)
True
```

`sage.misc.lazy_string.lazy_string(f, *args, **kwargs)`

Creates a lazy string.

**INPUT:**

- `f`, either a callable or a (format) string
- positional arguments that are given to `f`, either by calling or by applying it as a format string
- named arguments, that are forwarded to `f` if it is not a string

**EXAMPLES:**

```python
sage: from sage.misc.lazy_string import lazy_string
sage: f = lambda x: "laziness in "+str(x)
sage: s = lazy_string(f, ZZ); s
'1'laziness in Integer Ring'
```

```python
sage: from sage.all import *
>>> from sage.misc.lazy_string import lazy_string
>>> f = lambda x: "laziness in "+str(x)
>>> s = lazy_string(f, ZZ); s
'1'laziness in Integer Ring'
```

Here, we demonstrate that the evaluation is postponed until the value is needed, and that the result is not cached:

```python
sage: class C:
......:     def __repr__(self):
......:         print("determining string representation")
......:         return "a test"
sage: c = C()
sage: s = lazy_string("this is $s", c)
sage: s
determining string representation
l'this is a test'
sage: s == 'this is a test'
determining string representation
True
```

```python
>>> from sage.all import *
>>> class C:
...     def __repr__(self):
...         print("determining string representation")
...         return "a test"
>>> c = C()
>>> s = lazy_string("this is $s", c)
>>> s
determining string representation
l'this is a test'
>>> s == 'this is a test'
```

(continues on next page)
2.8 Caching

2.8.1 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.

EXAMPLES:

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

Python functions:

```python
sage: @cached_function
def test_pfunc(x):
    '''
    Some documentation
    '''
    return -x

sage: test_pfunc(5) is test_pfunc(5)
True
```

```python
>>> from sage.all import *

>>> @cached_function
def test_pfunc(x):
    '''
    Some documentation
    '''
    return -x

>>> test_pfunc(Integer(5)) is test_pfunc(Integer(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 Issue #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 CachedWeakValueDictionary for storing the results.
Cython cdef functions do not allow arbitrary decorators. However, one can wrap a Cython function and turn it into a cached function, by Issue #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: # needs sage.misc.cython
sage: cython('''cpdef test_funct(x): return -x''')
sage: wrapped_funct = cached_function(test_funct, name='wrapped_funct')
sage: wrapped_funct
Cached version of <cyfunction test_funct at ...>
sage: wrapped_funct.__name__
'wrapped_funct'
sage: wrapped_funct(5)
-5
```

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 Issue #11115, this is the case for all classes inheriting from Parent. See below for a more explicit example. By Issue #12951, cached methods of extension classes can be defined by simply using the decorator. However, an indirect approach is still needed for cpdef methods:

```
sage: # needs sage.misc.cython
cython_code = ['cpdef test_meth(self,x):
    "some doc for a wrapped cython method",
    return -x,
    'from sage.misc.cachefunc 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 <cyfunction MyClass.direct_method at ...>
sage: O.wrapped_method
Cached version of <cyfunction test_meth at ...>
sage: O.wrapped_method.__name__
wrapped_method
sage: O.wrapped_method(5)
-5
```

(continues on next page)
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 Issue #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 `CachedWeakValueDictionary` for storing the results.

By Issue #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):

```python
sage: cython_code = [
"from sage.misc.cachefunc import cached_method",
....: "from sage.misc.cachefunc import cached_in_parent_method",
....: "from sage.categories.category import Category",
....: "from sage.categories.objects import Objects",
....: "class MyCategory(Category):
....:   @cached_method",
....:   "def super_categories(self):
....:     return [Objects()]",
....:   "class ElementMethods:
....:     @cached_method",
....:     "def element_cache_test(self):
....:       pass",
]
```
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 be used.
Utilities, Release 10.4

(continued from previous page)

```python
....: " 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>$self.x'",
....: " def __hash__(self):
....: " return hash(self.x)",
....: " cpdef _richcmp_(left, right, int op):
....: " return PyObject_RichCompare(left.x, right.x, op)",
....: " def raw_test(self):
....: " return -self",
....: "cdef class MyBrokenElement(Element):",
....: " cdef public object x",
....: " def __init__(self,P,x):",
....: " self.x=x",
....: " Element.__init__(self,P)",
....: " def __neg__(self):
....: " return MyBrokenElement(self.parent(),-self.x)",
....: " def __repr__(self):
....: " return '<%s>$self.x'",
....: " def __hash__(self):
....: " return hash(self.x)",
....: " cpdef _richcmp_(left, right, int op):
....: " return PyObject_RichCompare(left.x, right.x, op)",
....: " 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)",
....: " cpdef _richcmp_(left, right, int op):
....: " return PyObject_RichCompare(left.x, right.x, op)",
....: " 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)

>>> from sage.all import *
```
... "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':%s
... " def __hash__(self):",
... " return hash(self.x)
... " cpdef _richcmp_(left, right, int op):",
... " return PyObject_RichCompare(left.x, right.x, op)
... " def raw_test(self):",
... " return -self",
... "from sage.structure.parent import Parent",
... "cdef class MyParent(Parent):",
... " Element = MyElement"

>>> cython('\n'.join(cython_code))
>>> P = MyParent(category=C)
>>> ebroken = MyBrokenElement(P, Integer(5))
>>> e = MyElement(P, Integer(5))

The cached methods inherited by the parent works:

```python
sage: # needs sage.misc.cython
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
```

```python
>>> from sage.all import *
>>> # needs sage.misc.cython
>>> P.one()
1
>>> P.one() is P.one()
True
>>> P.invert(e)
-5
>>> P.invert(e) is P.invert(e)
True
```

The cached methods inherited by MyElement works:

```python
sage: # needs sage.misc.cython
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  #...
       needs sage.misc.cython
True
sage: type(e) == type(ebroken)  #...
       needs sage.misc.cython
False
sage: ebroken.element_via_parent_test() is e.element_via_parent_test()  #...
       needs sage.misc.cython
True
```

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

```
sage: ebroken.element_cache_test()  #...
       needs sage.misc.cython
<5>
sage: ebroken.element_cache_test() is ebroken.element_cache_test()  #...
       needs sage.misc.cython
False
```

The cache can be emptied:

```
sage: # needs sage.misc.cython
sage: a = test_pfunc(5)
```
```python
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
```

```python
>>> from sage.all import *

# needs sage.misc.cython
>>> a = test_pfunc(Integer(5))
>>> test_pfunc.clear_cache()
>>> a is test_pfunc(Integer(5))
False
>>> a = P.one()
>>> P.one.clear_cache()
>>> 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:

```python
sage: b = ebroken.element_via_parent_test()  #--
# needs sage.misc.cython
sage: e.element_via_parent_test().clear_cache()  #--
# needs sage.misc.cython
sage: b is ebroken.element_via_parent_test()  #--
# needs sage.misc.cython
False
```

```python
>>> from sage.all import *

# needs sage.misc.cython
>>> b = ebroken.element_via_parent_test()  #--
# needs sage.misc.cython
>>> e.element_via_parent_test().clear_cache()  #--
# needs sage.misc.cython
>>> b is ebroken.element_via_parent_test()  #--
# needs sage.misc.cython
False
```

Introspection works:

```python
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
```

(continues on next page)
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: 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()  # _
needs sage.libs.singular
[a, b]
sage: I.groebner_basis() is I.groebner_basis()  #_
needs sage.libs.singular
True
sage: type(I.gens)
<class 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
sage: type(I.groebner_basis)
<class 'sage.misc.cachefunc.CachedMethodCaller'>
```
By Issue #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:

```python
sage: # needs sage.misc.cython
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"
]
```

```python
cython(os.linesep.join(cython_code))
```

```python
P = MyClass()
P.f(2, 3)
```

```python
6
```

```python
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:

```python
sage: # needs sage.misc.cython
cython_code = [  
... "from sage.misc.cachefunc import cached_method",
... "cdef class MyOtherClass:"
]
```

(continues on next page)
....: " cdef dict D",
....: " def __init__(self):
....: " self.D = {}",
....: " def __setattr__(self, n,v):
....: " self.D[n] = v",
....: " def __getattr__(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

>>> from sage.all import *

>>> # needs sage.misc.cython

>>> 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 __getattr__(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"
]

>>> cython(os.linesep.join(cython_code))

>>> Q = MyOtherClass()

>>> Q.f(Integer(2), Integer(3))
5

>>> Q.f(Integer(2), Integer(3)) is Q.f(Integer(2), Integer(3))
True

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

sage: timeit("a = P.f(2,3)")  # random
# needs sage.misc.cython
625 loops, best of 3: 1.3 µs per loop

sage: timeit("a = Q.f(2,3)")  # random
# needs sage.misc.cython
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:

```python
sage: # needs sage.rings.padics
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)
```

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`:

```python
sage: # needs sage.rings.padics
sage: hash(b)
Traceback (most recent call last):
  ...
TypeError: unhashable type: 'sage.rings.padics.qadic_flint_CR.qAdicCappedRelativeElement'
sage: from sage.misc.cachefunc import cached_method
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
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

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

```python
>>> from sage.all import *

# needs sage.rings.padics
>>> hash(b)
Traceback (most recent call last):
...
TypeError: unhashable type: 'sage.rings.padics.qadic_flint_CR.qAdicCappedRelativeElement'
>>> from sage.misc.cachefunc import cached_method
>>> @cached_method
... def f(x):
...     return x == a

>>> f(b)
True
>>> f(c)  # if b and c were hashable, this would return True
False
>>> b._cache_key()
(..., (0, 1),), 0, 1
>>> 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 \(a \neq b\), then also \(a._\text{cache_key()} \neq b._\text{cache_key()}\). In practice this means that the \_cache_key should always include the parent as its first argument:

```python
sage: S.<a> = Qq(4)       # needs sage.rings.padics
sage: d = a.add_bigoh(1)  # needs sage.rings.padics
sage: b._cache_key() == d._cache_key()  # this would be True if the parents were not included
False
```

```python
>>> from sage.all import *

# needs sage.rings.padics
>>> S = Qq(Integer(4), names=('a',)); (a,) = S._first_ngens(1)# needs sage.rings.padics
>>> d = a.add_bigoh(Integer(1))  # needs sage.rings.padics
>>> b._cache_key() == d._cache_key()  # this would be True if the parents were not included
False
```

```python
class sage.misc.cachefunc.CacheDict
    Bases: dict

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
- do_pickle – (optional boolean) whether or not the contents of the cache should be included when pickling this function; the default is not to include them.

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):

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

The inputs to the function must be hashable or they must define \texttt{sage}\_structure\_sage\_object.\_SageObject\_.\_cache\_key().

EXAMPLES:

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

>>> from sage.all import *
>>> @cached_function
... def mul(x, y=\texttt{Integer}(2)):
...     return x*y
>>> mul(\texttt{Integer}(3))
6
```

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

```python
sage: mul(3) \texttt{is} mul(3,2)
True
sage: mul(3,y=2) \texttt{is} mul(3,2)
True

>>> from sage.all import *
>>> mul(\texttt{Integer}(3)) \texttt{is} mul(\texttt{Integer}(3),\texttt{Integer}(2))
True
>>> mul(\texttt{Integer}(3),y=\texttt{Integer}(2)) \texttt{is} mul(\texttt{Integer}(3),\texttt{Integer}(2))
True
```

The user can clear the cache:

```python
sage: a = mul(4)
sage: mul.clear\_cache()
sage: a \texttt{is} mul(4)
False
```
```python
>>> from sage.all import *
>>> a = mul(Integer(4))
>>> mul.clear_cache()
>>> a is mul(Integer(4))
False
```

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

```python
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`:

```python
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,'default')
True
```

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

```python
>>> from sage.all import *
>>> @cached_function(*args, **kwds)
... def f(x):
...     return x
>>> f.cached(5)
Traceback (most recent call last):
... KeyError: ((5,), ())
>>> f(5)
5
>>> f.cached(5)
5
```

**cache**

**cached(*args, **kwds)**

Return the result from the cache if available. If the value is not cached, raise `KeyError`.

**EXAMPLES:**

```python
sage: @cached_function
... def f(x):
...     return x
sage: f.cached(5)
Traceback (most recent call last):
... KeyError: ((5,), ())
sage: f(5)
5
sage: f.cached(5)
5
```
Utilities, Release 10.4

```python
>>> from sage.all import *
>>> @cached_function
... def f(x):
...     return x

>>> f.cached(Integer(5))
Traceback (most recent call last):
  ...
KeyError: ((5,), ())
>>> f(Integer(5))
5
>>> f.cached(Integer(5))
5
```

clear_cache()

Clear the cache dictionary.

EXAMPLES:

```
sage: # needs sage.combinat
g = CachedFunction(number_of_partitions)
sage: a = g(5)  # needs sage.libs.flint

→ needs sage.libs.flint
sage: g.cache  # needs sage.libs.flint
{((5, 'default'), ()): 7}
sage: g.clear_cache()
sage: g.cache
{}
```

```python
>>> from sage.all import *
>>> # needs sage.combinat
g = CachedFunction(number_of_partitions)
>>> a = g(Integer(5))  # needs sage.libs.flint

→ needs sage.libs.flint{((5, 'default'), ()): 7}
>>> g.clear_cache()
>>> g.cache
{}
```

f
g.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,), ())
```

2.8. Caching  277
>>> from sage.all import *
>>> @cached_function
... def foo(x):
...     return x**Integer(2)
>>> foo(Integer(2))
4
>>> foo.get_key(Integer(2))
((2,), ())
>>> foo.get_key(x=Integer(3))
((3,), ())

Examples for cached methods:

```
sage: class Foo:
    ....: def __init__(self, x):
    ....:     self._x = x
    ....: @cached_method
    ....: def f(self, y, z=Integer(0)):
    ....:     return self._x * y + z

sage: a = Foo(2)
sage: z = a.f(37)
sage: k = a.f.get_key(Integer(37)); k
((37, 0), ())
sage: a.f.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=Integer(6))
((1, 2, 3), (('x', 4), ('y', 5), ('z', 6)))
```

```
>>> from sage.all import *
>>> class Foo:
    ...    def __init__(self, x):
    ...        self._x = x
    ...    @cached_method
    ...    def f(self, y, z=Integer(0)):
    ...        return self._x * y + z

>>> a = Foo(Integer(2))
>>> z = a.f(Integer(37))
>>> k = a.f.get_key(Integer(37)); k
((37, 0), ())
>>> a.f.cache[k] is z
True
```

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)
```

(continues on next page)
Utilities, Release 10.4

is_in_cache(*args, **kwds)
Checks if the argument list is in the cache.

EXAMPLES:

```python
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
```

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:

```python
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
```

```python
>>> from sage.all import *
>>> @cached_function
def oddprime_factors(n):
    l = [p for p, e in factor(n) if p != Integer(2)]
    return len(l)
>>> oddprime_factors.precompute(range(Integer(1),Integer(100)), Integer(4))
>>> oddprime_factors.cache[(Integer(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:

```python
sage: # needs sage.combinat sage.libs.flint
sage: g = CachedFunction(number_of_partitions)
sage: a = g(5)
sage: g.cache
{((5, 'default'), ()): 7}
sage: g.set_cache(17, 5)
sage: g.cache
{((5, 'default'), ()): 17}
sage: g(5)
17
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.libs.flint
>>> g = CachedFunction(number_of_partitions)
>>> a = g(Integer(5))
>>> g.cache
{((5, 'default'), ()): 7}
>>> g.set_cache(Integer(17), Integer(5))
>>> g.cache
{((5, 'default'), ()): 17}
>>> g(Integer(5))
17
```

```python
class sage.misc.cachefunc.CachedInParentMethod
    Bases: 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`.

```python
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:**

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

```python
>>> from sage.all import *
>>> class Foo():
...     @cached_method
...     def f(self, t, x=Integer(2)):
...         print('computing')
...         return t**x
>>> a = Foo()
```

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

```python
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:

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

```python
sage: class A():
    ...
    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
```

The parameter `do_pickle` can be used to enable pickling of the cache. Usually the cache is not stored when pickling:

```python
sage: class A():
    ...
    @cached_method
    ...
    def f(self, x):
        return None
sage: import __main__
sage: __main__.A = A
sage: a = A()
sage: a.f(1)
```

(continues on next page)
sage: b = loads(dumps(a))
sage: len(b.f.cache)
0

>>> from sage.all import *
>>> class A():
...    @cached_method
...    def f(self, x):
...        return None

>>> import __main__
>>>
__main__.A = A
>>>
>>> a = A()
>>> a.f(Integer(1))
>>> len(a.f.cache)
1
>>> b = loads(dumps(a))
>>> len(b.f.cache)
1

When do_pickle is set, the pickle contains the contents of the cache:

sage: class A():
......    @cached_method(do_pickle=True)
......    def f(self, x):
......        return None

sage: __main__.A = A
sage: a = A()
sage: a.f(1)
sage: len(a.f.cache)
1
sage: b = loads(dumps(a))
sage: len(b.f.cache)
1

Cached methods cannot be copied like usual methods, see Issue #12603. Copying them can lead to very surprising results:

sage: class A:
......    @cached_method
......    def f(self):
......        return 1

sage: class B:
......    g=A.f
......    def f(self):

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

....:        return 2

sage: b=B()
sage: b.f()
2
sage: b.g()
1
sage: b.f()
1

>>> from sage.all import *

>>> class A:
...   @cached_method
...   def f(self):
...     return Integer(1)

>>> class B:
...   g=A.f
...   def f(self):
...     return Integer(2)

>>> b=B()
>>> b.f()
2
>>> b.g()
1
>>> b.f()
1

class sage.misc.cachefunc.CachedMethodCaller

Bases: CachedFunction

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

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

EXAMPLES:

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)
<class 'sage.misc.cachefunc.CachedMethodCaller'>
sage: a.bar(2) is a.bar(x=2)
True

>>> from sage.all import *

>>> class A:
...   @cached_method
...   def bar(self,x):

(continues on next page)
... return x**Integer(2)
>>> a = A()
>>> a.bar
Cached version of <function ...bar at 0x...>
>>> type(a.bar)
<class 'sage.misc.cachefunc.CachedMethodCaller'>
>>> a.bar(Integer(2)) is a.bar(x=Integer(2))
True

cached (*args, **kwds)

Return the result from the cache if available. If the value is not cached, raise KeyError.

EXAMPLES:

>>> from sage.all import *
>>> class CachedMethodTest():
... @cached_method
... def f(self, x):
... return x
... o = CachedMethodTest()
... CachedMethodTest.f.cached(o, 5)
Traceback (most recent call last):...
  ... KeyError: ((5,), ())
... o.f.cached(5)
Traceback (most recent call last):...
 ... KeyError: ((5,), ())
... o.f(5)
5
... CachedMethodTest.f.cached(o, 5)
5
... o.f.cached(5)
5

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:**

```python
sage: class Foo():
    ....:     @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}
True
```

```python
>>> from sage.all import *
>>> class Foo():
...     @cached_method
...     def f(self, i):
...         return i**Integer(2)

>>> foo = Foo()
>>> foo.f(Integer(3))
9
>>> foo.f(Integer(1))
1
>>> foo.f.precompute(range(Integer(2)), Integer(2))
>>> foo.f.cache == {((Integer(0),), ()): Integer(0), ((Integer(1),), ()): Integer(1), ((Integer(3),), ()): Integer(9)}
True
```

class `sage.misc.cachefunc.CachedMethodCallerNoArgs`

Bases: `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.

**EXAMPLES:**

```python
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)
<class 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
sage: I.gens is I.gens
True
```

(continues on next page)
AUTHOR:

• Simon King (2011-04)

clear_cache()

Clear the cache dictionary.

EXAMPLES:

```python
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:

```python
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 cannot cache the return value None.

EXAMPLES:

```python
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
```

```python
>>> from sage.all import *
>>>
P = QQ['x', 'y']; (x, y,) = P._first_ngens(2)
>>> I = P*[x,y]
>>> I.gens.is_in_cache()
False
>>> I.gens()
[x, y]
>>> 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:

```python
sage: P.<a,b,c,d> = QQ[]
sage: I = P*[a,b]
```

```python
sage: I.gens()
[a, b]
sage: I.gens.set_cache('bar')
sage: I.gens()
'bar'
```

```python
>>> from sage.all import *
>>>
P = QQ['a, b, c, d']; (a, b, c, d,) = P._first_ngens(4)
>>> I = P*[a,b]
>>> I.gens()
[a, b]
>>> I.gens.set_cache('bar')
>>> 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:

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

```python
>>> from sage.all import *
```
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:

```python
sage: J.groebner_basis.is_in_cache()  #...
   ← needs sage.libs.singular
True
sage: J.groebner_basis
Cached version of <function ...groebner_basis at 0x...>
sage: J.groebner_basis() == I.groebner_basis()  #...
   ← needs sage.libs.singular
True
```

AUTHOR:
• Simon King (2011-01)

```python
class sage.misc.cachefunc.CachedSpecialMethod
Bases: 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:
```
Utilities, Release 10.4

(continued from previous page)

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

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

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

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

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

```python
class sage.misc.cachefunc.DiskCachedFunction(f, dir, memory_cache=False, key=None)
Bases: CachedFunction
```

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

**EXAMPLES:**

```python
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
```

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

(continues on next page)
class sage.misc.cachefunc.FileCache(dir, prefix='', memory_cache=False)

Bases: object

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 \(\text{memory}_\text{cache} = \text{True}\), 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:

- \(\text{prefix-argstring.key.sobj}\) contains the key only,
- \(\text{prefix-argstring.sobj}\) contains the tuple \((\text{key, val})\)

where \(\text{self}[\text{key}] == \text{val}\).

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

clear()

Clear all key, value pairs from self and unlink the associated files from the file cache.

EXAMPLES:

```
sage: from sage.misc.cachefunc import FileCache
sage: dir = tmp_dir()
sage: FC1 = FileCache(dir, memory_cache=False, prefix='foo')
sage: FC2 = FileCache(dir, memory_cache=False, prefix='foo')
sage: k1 = ((), ((a, 1),))
sage: t1 = randint(0, 1000)
sage: k2 = ((), ((b, 1),))
sage: t2 = randint(0, 1000)
sage: FC1[k1] = t1
sage: FC2[k2] = t2
sage: FC2.clear()
sage: FC1[k1] == t1
True
sage: FC2[k2] == t2
True
```

```
>>> from sage.all import *
>>> from sage.misc.cachefunc import FileCache
>>> dir = tmp_dir()
>>> FC1 = FileCache(dir, memory_cache=False, prefix='foo')
>>> FC2 = FileCache(dir, memory_cache=False, prefix='foo')
>>> k1 = ((), (('a', Integer(1)),))
>>> t1 = randint(Integer(0), Integer(1000))
>>> k2 = ((), (('b', Integer(1)),))
```
Utilities, Release 10.4

(continued from previous page)

```python
>>> t2 = randint(Integer(0), Integer(1000))
>>> FC1[k1] = t1
>>> FC2[k2] = t2
>>> FC2.clear()
>>> k1 in FC1
False
>>> k2 in FC1
False
```

**file_list()**

Return the list of files corresponding to self.

**EXAMPLES:**

```python
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
```

```python
>>> from sage.all import *
>>> from sage.misc.cachefunc import FileCache

>>> dir = tmp_dir()

>>> FC = FileCache(dir, memory_cache = True, prefix='t')

>>> FC[(),()] = Integer(1)
>>> FC[({1,2},{}),] = Integer(2)
>>> FC[({1},({'a',1},)),] = Integer(3)

>>> 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:**

```python
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(); I
[(((), ()), 1), (((1), ('a', 1)),), 3, (((1, 2), ()), 2)]
```
Utilities, Release 10.4

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

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

EXAMPLES:

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

values()
Return a list of values that are stored in self.

EXAMPLES:

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

```
class sage.misc.cachefunc.GloballyCachedMethodCaller
Bases: CachedMethodCaller

Implementation of cached methods in case that the cache is not stored in the instance, but in some global object. In particular, it is used to implement CachedInParentMethod.

The only difference is that the instance is used as part of the key.

class sage.misc.cachefunc.NonpicklingDict
Bases: dict

A special dict which does not pickle its contents.

EXAMPLES:

sage: from sage.misc.cachefunc import NonpicklingDict
sage: d = NonpicklingDict()

>>> d[0] = 0

>>> loads(dumps(d))
{}

>>> from sage.all import *

>>> from sage.misc.cachefunc import NonpicklingDict

>>> d = NonpicklingDict()

>>> d[Integer(0)] = Integer(0)

>>> loads(dumps(d))
{}

class sage.misc.cachefunc.WeakCachedFunction
Bases: 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):

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

As an exception meant to improve performance, the most recently computed values are strongly referenced. The number of strongly cached values can be controlled by the `cache` keyword.

EXAMPLES:

```python
sage: from sage.misc.cachefunc import weak_cached_function
sage: class A: pass
sage: @weak_cached_function(cache=0)
    def f():
        ....: print("doing a computation")
```

(continues on next page)
return A()

doing a computation

>>> from sage.all import *
>>> from sage.misc.cachefunc import weak_cached_function
>>> class A: pass
>>> @weak Cached function(cache=Integer(0))
... def f():
...   print("doing a computation")
...   return A()

The result is cached:

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

However, if there are no strong references left, the result is deleted, and thus a new computation takes place:

sage: del a
sage: del b
sage: del a
sage: del b

Above, we used the cache=0 keyword. With a larger value, the most recently computed values are cached anyway, even if they are not referenced:

sage: @weak_cached_function(cache=3)
....: def f(x):
....:   print("doing a computation for x={}".format(x))
....:   return A()
sage: a = f(1); del a
doing a computation for x=1
sage: a = f(2), f(1); del a
doing a computation for x=2
sage: a = f(3), f(1); del a
doing a computation for x=3
sage: a = f(4), f(1); del a
doing a computation for x=4
doing a computation for x=1
sage: a = f(5), f(1); del a
doing a computation for x=5
The parameter key can be used to ignore parameters for caching. In this example we ignore the parameter algorithm:

```python
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")
True
```

```

sage.misc.cachefunc.cache_key(o)
```

Helper function to return a hashable key for `o` which can be used for caching.

This function is intended for objects which are not hashable such as $p$-adic numbers. The difference from calling an object’s `cache_key` method directly, is that it also works for tuples and unpacks them recursively (if necessary, i.e., if they are not hashable).

**EXAMPLES:**

```python
sage: from sage.misc.cachefunc import cache_key
sage: K.<u> = Qq(9)  # needs sage.rings.padics
sage: a = K(1); a  # needs sage.rings.padics
1 + O(3^20)
sage: cache_key(a)  # needs sage.rings.padics
(..., ((1,),), 0, 20)
```

```python
>>> from sage.all import *
>>> from sage.misc.cachefunc import cache_key
```
This function works if \( o \) is a tuple. In this case it unpacks its entries recursively:

```
sage: o = (1, 2, (3, a))  #...
    # needs sage.rings.padics
sage: cache_key(o)  #...
    # needs sage.rings.padics
(1, 2, (3, (...), (1,)), 0, 20))
```

Note that tuples are only partially unpacked if some of its entries are hashable:

```
sage: o = (1/2, a)  #...
    # needs sage.rings.padics
sage: cache_key(o)  #...
    # needs sage.rings.padics
(1/2, (...), (1,)), 0, 20))
```

```
sage: from sage.all import *

>>> o = (Integer(1)/Integer(2), a)
    # needs sage.rings.padics
>>> cache_key(o)
    # needs sage.rings.padics
(1/2, (...), (1,)), 0, 20))
```

`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
- \( \text{name} \) – (optional string) name that the cached version of \( f \) should be provided with
- \( \text{key} \) – (optional callable) takes the input and returns a key for the cache, typically one would use this to normalize input
- \( \text{do_pickle} \) – (optional boolean) whether or not the contents of the cache should be included when pickling this function; the default is not to include them.

If \( f \) is a function, do either \( g = \text{CachedFunction}(f) \) or \( g = \text{cached_function}(f) \) to make a cached version of \( f \), or put `@cached_function` right before the definition of \( f \) (i.e., use Python decorators):
The inputs to the function must be hashable or they must define `sage.structure.sage_object.SageObject._cache_key()`.

**EXAMPLES:**

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

```python
>>> from sage.all import *
>>> @cached_function
... def mul(x, y=Integer(2)):
...     return x*y
>>> mul(Integer(3))
6
```

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

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

```python
>>> from sage.all import *
>>> mul(Integer(3)) is mul(Integer(3),Integer(2))
True
>>> mul(Integer(3),y=Integer(2)) is mul(Integer(3),Integer(2))
True
```

The user can clear the cache:

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

```python
>>> from sage.all import *
>>> a = mul(Integer(4))
>>> mul.clear_cache()
>>> a is mul(Integer(4))
False
```

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

```python
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:

```python
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)
True
```

```
from sage.all import *
>>> mul.set_cache('foo',Integer(5))
>>> mul(Integer(5),Integer(2))
'foo'
```

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

```python
sage: @cached_function(key=lambda x,y,algorithm: (x,y))
    def mul(x, y, algorithm="default"):
        return x*y
```

```
>>> from sage.all import *
>>> @cached_function(key=lambda x,y,algorithm: (x,y))
    def mul(x, y, algorithm="default"):
        return x*y
```
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: c = C()
sage: type(C.__hash__)
<class 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
sage: hash(c)
compute hash
5
```

```
>>> from sage.all import *
>>> class C:
...     @cached_method
...     def __hash__(self):
...         print("compute hash")
...         return int(Integer(5))
...     @cached_method
...     def f(self, x):
...         print("computing cached method")
...         return x*Integer(2)

>>> hash(c)
compute hash
5
```

Different instances have distinct caches:

```
sage: d = C()
sage: d.f is c.f
False
```

(continues on next page)
sage: c.f(4)
8
sage: d.f(4)
computing cached method
8

>>> from sage.all import *
>>> d = C()
>>> d.f(Integer(4)) is c.f(Integer(4))
computing cached method
False
>>> d.f.clear_cache()
>>> c.f(Integer(4))
8
>>> d.f(Integer(4))
computing cached method
8

Using cached methods for the hash and other special methods was implemented in Issue #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 ...>

The parameter `do_pickle` can be used if the contents of the cache should be stored in a pickle of the cached method. This can be dangerous with special methods such as `__hash__`:

```python
sage: class C:
    ....:     @cached_method(do_pickle=True)
    ....:     def __hash__(self):
    ....:         return id(self)
```

```python
>>> import __main__
>>> C = __main__.C
>>> c = C()
>>> hash(c)  # random output
>>> d = loads(dumps(c))
>>> hash(d) == hash(c)
True
```

(continues on next page)
However, the contents of a method’s cache are not pickled unless do_pickle is set:

```
>>> from sage.all import *
>>> class C:
...     @cached_method
...     def __hash__(self):
...         return id(self)

>>> __main__.C = C
>>> c = C()
>>> hash(c)  # random output
>>> d = loads(dumps(c))
>>> hash(d) == hash(c)
False
```

```
sage.misc.cachefunc.dict_key(o)

Return a key to cache object o in a dict.

This is different from cache_key since the cache_key might get confused with the key of a hashable object. Therefore, such keys include unhashable_key which acts as a unique marker which is certainly not stored in the dictionary otherwise.

EXAMPLES:
```
```
```
```
>>> K = Qq(Integer(9), names=('u',)); (u,) = K._first_ngens(1)# needs sage.rings.padics
>>> dict_key(u)
# needs sage.rings.padics
<Object object at ...>, (...), 20)

```python
class sage.misc.cachefunc.disk_cached_function(dir, memory_cache=False, key=None)
Bases: object

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
# needs sage.libs.pari
11
sage: @disk_cached_function(dir)
....: def foo(x):
    return 1/x
sage: foo(200)
# needs sage.libs.pari
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):

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

As an exception meant to improve performance, the most recently computed values are strongly referenced. The
number of strongly cached values can be controlled by the cache keyword.

EXAMPLES:
Utilities, Release 10.4

sage: from sage.misc.cachefunc import weak_cached_function
sage: class A: pass
sage: @weak_cached_function(cache=0)
....: 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 is deleted, and thus a new computation takes place:

sage: del a
sage: del b
sage: a = f()
doing a computation

Above, we used the cache=0 keyword. With a larger value, the most recently computed values are cached anyway, even if they are not referenced:

sage: @weak_cached_function(cache=3)
....: def f(x):
....:     print("doing a computation for x={}".format(x))
....:     return A()

sage: a = f(1); del a
doing a computation for x=1
sage: a = f(2), f(1); del a
doing a computation for x=2
sage: a = f(3), f(1); del a
doing a computation for x=3

(continues on next page)
The parameter `key` can be used to ignore parameters for caching. In this example we ignore the parameter `algorithm`:

```python
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(Integer(1),algorithm="algorithm")
True
```

2.8.2 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:
Utilities, Release 10.4

```python
sage: import weakref
sage: class Vals:
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()
....: def __ne__(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

sage: len(D) > 1
True
```

Hence, 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:

```python
>>> from sage.all import *
>>> import weakref
>>> class Vals:
pass
>>> 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()
...     def __ne__(self, other):
...         return self.val() != other.val()

>>> ValList = [Vals() for _ in range(Integer(10))]

>>> D = weakref.WeakValueDictionary()

>>> for v in ValList:
...     D[Keys(v)] = v

>>> len(D)
10

>>> del ValList, v

>>> len(D) > Integer(1)
True
```

2.8. Caching
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`:

```python
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.
The variant `CachedWeakValueDictionary` additionally adds strong references to the most recently added values. This ensures that values will not be immediately deleted after adding them to the dictionary. This is mostly useful to implement cached functions.

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

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

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

```python
class sage.misc.weak_dict.CachedWeakValueDictionary
    Bases: WeakValueDictionary

    This class extends `WeakValueDictionary` with a strong cache to the most recently added values. It is meant to solve the case where significant performance losses can occur if a value is deleted too early, but where keeping a value alive too long does not hurt much. This is typically the case with cached functions.
```

**EXAMPLES:**

We illustrate the difference between `WeakValueDictionary` and `CachedWeakValueDictionary`. An item is removed from a `WeakValueDictionary` as soon as there are no references to it:

```python
sage: from sage.misc.weak_dict import WeakValueDictionary
sage: D = WeakValueDictionary()
sage: class Test(): pass
t sage: tmp = Test()
sage: D[0] = tmp
sage: 0 in D
True
sage: del tmp
sage: 0 in D
False
```

```python
>>> from sage.all import *
>>> from sage.misc.weak_dict import WeakValueDictionary
>>> D = WeakValueDictionary()
>>> class Test(): pass
>>> tmp = Test()
>>> D[Integer(0)] = tmp
>>> Integer(0) in D
True
>>> del tmp
>>> Integer(0) in D
False
```

So, if you have a cached function repeatedly creating the same temporary object and deleting it (in a helper function called from a loop for example), this caching will not help at all. With `CachedWeakValueDictionary`, the most recently added values are not deleted. After adding enough new values, the item is removed anyway:
```python
sage: from sage.misc.weak_dict import CachedWeakValueDictionary
sage: D = CachedWeakValueDictionary(cache=4)
sage: class Test(): pass
sage: tmp = Test()
sage: D[0] = tmp
sage: 0 in D
True
sage: del tmp
sage: 0 in D
True
sage: for i in range(5):
    ...:     D[i] = Test()
    ...:     print(0 in D)
True
True
True
False
False
```

```python
>>> from sage.all import *
>>> from sage.misc.weak_dict import CachedWeakValueDictionary
>>> D = CachedWeakValueDictionary(cache=Integer(4))
>>> class Test(): pass
>>> tmp = Test()
>>> D[Integer(0)] = tmp
>>> Integer(0) in D
True
>>> del tmp
>>> Integer(0) in D
True
>>> for i in range(Integer(5)):
    ...:     D[Integer(1)] = Test()
    ...:     print(Integer(0) in D)
True
True
True
False
False
```

```python
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:
```
```
```python
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
```
Utilities, Release 10.4

>>> from sage.all import *
>>> from sage.misc.weak_dict import WeakValueDictionary
>>> v = frozenset([Integer(1)])
>>> D = WeakValueDictionary({Integer(1) : v})
>>> len(D)
1
>>> del v
>>> 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
go
sage: class Vals():
    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()
    ....:    def __ne__(self, other):
    ....:        return self.val() != other.val()

sage: ValList = [Vals() for _ in range(10)]

sage: import sage.misc.weak_dict
go
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

>>> from sage.all import *
>>> import weakref
go
>>> class Vals():
    pass
(continues on next page)
>>> 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()
...     def __ne__(self, other):
...         return self.val() != other.val()

>>> ValList = [Vals() for _ in range(Integer(10))]

>>> import sage.misc.weak_dict

>>> D = sage.misc.weak_dict.WeakValueDictionary()

>>> for v in ValList:
...     D[Keys(v)] = v

>>> len(D)
10

>>> del ValList

>>> len(D)
1

>>> del v

>>> len(D)
0

def get(k, d=None)

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

The default value defaults to None.

EXAMPLES:

sage: import sage.misc.weak_dict

sage: # needs sage.libs.pari

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

>>> from sage.all import *

>>> import sage.misc.weak_dict

>>> # needs sage.libs.pari

>>> L = [GF(p) for p in prime_range(Integer(10)**Integer(3))]  

>>> D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))

>>> Integer(100) in D
True

>>> Integer(200) in D
False

>>> D.get(Integer(100), "not found")
Finite Field of size 547

```python
>>> D.get(Integer(200), "not found")
'not found'
>>> D.get(Integer(200)) is None
True
```

**items()**

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:**

```python
sage: import sage.misc.weak_dict
sage: class Vals:
    ....:     def __init__(self, n):
    ....:         self.n = n
    ....:     def __repr__(self):
    ....:         return "<%s>" % self.n
    ....:     def __lt__(self, other):
    ....:         return self.n < other.n
    ....:     def __eq__(self, other):
    ....:         return self.n == other.n
    ....:     def __ne__(self, other):
    ....:         return self.val() != other.val()

sage: class Keys():
    ....:     def __init__(self, n):
    ....:         self.n = n
    ....:     def __hash__(self):
    ....:         if self.n % 2:
    ....:             return int(5)
    ....:         return int(3)
    ....:     def __repr__(self):
    ....:         return "[%s]" % self.n
    ....:     def __lt__(self, other):
    ....:         return self.n < other.n
    ....:     def __eq__(self, other):
    ....:         return self.n == other.n
    ....:     def __ne__(self, other):
    ....:         return self.val() != other.val()

sage: L = [(keys(n), vals(n)) for n in range(10)]
```

```python
sage: from sage.all import *
>>> import sage.misc.weak_dict
>>> class Vals:
...     def __init__(self, n):
...         self.n = n
...     def __repr__(self):
...         return "<%s>" % self.n
...     def __lt__(self, other):
...         return self.n < other.n
...     def __eq__(self, other):
...         return self.n == other.n
...     def __ne__(self, other):
...         return self.val() != other.val()
```
...     return self.n == other.n
... def __ne__(self, other):
...     return self.val() != other.val()

>>> class Keys():
...     def __init__(self, n):
...         self.n = n
...     def __hash__(self):
...         if self.n % Integer(2):
...             return int(Integer(5))
...         return int(Integer(3))
...     def __repr__(self):
...         return "[{}]".format(self.n)
...     def __lt__(self, other):
...         return self.n < other.n
...     def __eq__(self, other):
...         return self.n == other.n
...     def __ne__(self, other):
...         return self.val() != other.val()

L = [(Keys(n), Vals(n)) for n in range(Integer(10))]
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.items()):
....:     print("{} {}\n[0] <0>
[1] <1>
[3] <3>
[4] <4>
[6] <6>
[7] <7>
[8] <8>
[9] <9>

>>> from sage.all import *
>>> del D[Keys(Integer(2))]
>>> del L[Integer(5)]
>>> for k,v in sorted(D.items()):
....:     print("{} {}\n[0] <0>
[1] <1>
[3] <3>
[4] <4>
[6] <6>
[7] <7>
[8] <8>
[9] <9>

items_list()
The key-value pairs of this dictionary.

EXAMPLES:
```python
# Utilities, Release 10.4

sage: import sage.misc.weak_dict
sage: class Vals:
    ... def __init__(self, n):
    ...     self.n = n
    ... def __repr__(self):
    ...     return "<%s" % self.n
    ... def __lt__(self, other):
    ...     return self.n < other.n
    ... def __eq__(self, other):
    ...     return self.n == other.n
    ... def __ne__(self, other):
    ...     return self.val() != other.val()

sage: class Keys():
    ... def __init__(self, n):
    ...     self.n = n
    ... def __hash__(self):
    ...     if self.n % 2:
    ...         return int(5)
    ...     return int(3)
    ... def __repr__(self):
    ...     return "[%s]" % self.n
    ... def __lt__(self, other):
    ...     return self.n < other.n
    ... def __eq__(self, other):
    ...     return self.n == other.n
    ... def __ne__(self, other):
    ...     return self.val() != other.val()

sage: L = [(Keys(n), Vals(n)) for n in range(10)]
sage: D = sage.misc.weak_dict.WeakValueDictionary(L)

>>> from sage.all import *
>>> import sage.misc.weak_dict
>>> class Vals:
...     def __init__(self, n):
...         self.n = n
...     def __repr__(self):
...         return "<%s" % self.n
...     def __lt__(self, other):
...         return self.n < other.n
...     def __eq__(self, other):
...         return self.n == other.n
...     def __ne__(self, other):
...         return self.val() != other.val()

>>> class Keys():
...     def __init__(self, n):
...         self.n = n
...     def __hash__(self):
...         if self.n % Integer(2):
...             return int(Integer(5))
...         return int(Integer(3))
...     def __repr__(self):
...         return "[%s]" % self.n
...     def __lt__(self, other):
...         return self.n < other.n
...     def __eq__(self, other):
...         return self.n == other.n
...     def __ne__(self, other):
...         return self.val() != other.val()

(continues on next page)
```
... return self.val() != other.val()

>>> L = [(Keys(n), Vals(n)) for n in range(Integer(10))]
>>> 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\>)]
```

```
>>> from sage.all import *
>>> del D[Keys(Integer(2))]
>>> del L[Integer(5)]
```
```
>>> sorted(D.items())
[(\[0\], \<0\>),
 (\[1\], \<1\>),
 (\[3\], \<3\>),
 (\[4\], \<4\>),
 (\[6\], \<6\>),
 (\[7\], \<7\>),
 (\[8\], \<8\>),
 (\[9\], \<9\>)]
```

**iteritems()**

**EXAMPLES:**

```
sage: import sage.misc.weak_dict
sage: class Vals(): pass
sage: L = [Vals() for _ in range(10)]
sage: D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))
sage: T = list(D.iteritems())
doctest:warning...:
DeprecationWarning: use items instead
See https://github.com/sagemath/sage/issues/34488 for details.
```

```
>>> from sage.all import *
>>> import sage.misc.weak_dict
>>> class Vals(): pass
```
```
>>> L = [Vals() for _ in range(Integer(10))]
>>> D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))
>>> T = list(D.iteritems())
doctest:warning...:
DeprecationWarning: use items instead
See https://github.com/sagemath/sage/issues/34488 for details.
```

**itervalues()**

Deprecated.
EXAMPLES:

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

```python
sage: D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))
```

```python
doctest:warning...:
DeprecationWarning: use values instead
See https://github.com/sagemath/sage/issues/34488 for details.
```

```python
>>> from sage.all import *
>>> import sage.misc.weak_dict
>>> class Vals(): pass

>>> L = [Vals() for _ in range(Integer(10))]
```

```python
>>> D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))
```

```python
>>> del L[4]
```

One item got deleted from the list \texttt{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:

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

```python
>>> from sage.all import *

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

\texttt{keys()} 

The list of keys.

EXAMPLES:

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

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

```python
>>> from sage.all import *
>>> import sage.misc.weak_dict
```

```python
>>> class Vals(): pass

>>> L = [Vals() for _ in range(Integer(10))]
```

```python
>>> D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))
```

```python
>>> del L[Integer(4)]
```

One item got deleted from the list \texttt{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:

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

```python
>>> from sage.all import *

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

\texttt{pop}(k) 

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

EXAMPLES:

```python
sage: import sage.misc.weak_dict
sage: # needs sage.libs.pari
```
Utilities, Release 10.4

(continued from previous page)

```sage
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
```

```python
>>> from sage.all import *
>>> import sage.misc.weak_dict

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

**popitem()**

Return and delete some item from the dictionary.

**EXAMPLES:**

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

```python
>>> from sage.all import *
>>> import sage.misc.weak_dict

>>> D = sage.misc.weak_dict.WeakValueDictionary()
>>> D[Integer(1)] = ZZ
```

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

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

```python
>>> from sage.all import *
>>> 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'

>>> from sage.all import *
>>> D.popitem()
Traceback (most recent call last):
...
KeyError: 'popitem(): weak value dictionary is empty'

setdefault\((k, default=\text{None})\)

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: # needs sage.libs.pari
sage: L = [(p, GF(p)) for p in prime_range(10)]

sage: D = sage.misc.weak_dict.WeakValueDictionary(L)

sage: len(D)
4

>>> from sage.all import *

>>> import sage.misc.weak_dict

>>> # needs sage.libs.pari

>>> L = [(p, GF(p)) for p in prime_range(Integer(10))]

>>> D = sage.misc.weak_dict.WeakValueDictionary(L)

>>> len(D)
4

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

sage: # needs sage.libs.pari
sage: D.setdefault(5, ZZ)
Finite Field of size 5

sage: D[5]
Finite Field of size 5

>>> from sage.all import *

>>> # needs sage.libs.pari

>>> D.setdefault(Integer(5), ZZ)
Finite Field of size 5

>>> D[Integer(5)]
Finite Field of size 5

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

sage: # needs sage.libs.pari
sage: 4 in D
False

sage: D.setdefault(4, ZZ)
Integer Ring

sage: 4 in D
(True, continues on next page)
True

```python
>>> sage: D[4]
Integer Ring
>>> sage: len(D)
5
```

```python
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> Integer(4) in D
False
>>> D.setdefault(Integer(4), ZZ)
Integer Ring
>>> Integer(4) in D
True
>>> D[Integer(4)]
Integer Ring
>>> len(D)
5
```

```python
values()
```
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:**

```python
sage: import sage.misc.weak_dict
sage: class Vals:
    ... def __init__(self, n):
    ...     self.n = n
    ... def __repr__(self):
    ...     return "<\%s>" % self.n
    ... def __lt__(self, other):
    ...     return self.n < other.n
    ... def __eq__(self, other):
    ...     return self.n == other.n
    ... def __ne__(self, other):
    ...     return self.val() != other.val()
    ... def __lt__(self, other):
    ...     return self.n < other.n
    ... def __eq__(self, other):
    ...     return self.n == other.n
    ... def __ne__(self, other):
    ...     return self.val() != other.val()
```

```python
sage: L = [Vals(n) for n in range(10)]
sage: D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))
```

```python
>>> from sage.all import *
>>> import sage.misc.weak_dict
>>> class Vals:
    ... def __init__(self, n):
    ...     self.n = n
    ... def __repr__(self):
    ...     return "<\%s>" % self.n
    ... def __lt__(self, other):
    ...     return self.n < other.n
    ... def __eq__(self, other):
    ...     return self.n == other.n
    ... def __ne__(self, other):
    ...     return self.val() != other.val()
```

(continues on next page)
return self.val() != other.val()

>>> L = [Vals(n) for n in range(Integer(10))]

>>> 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:

\[
\text{sage: } \text{del } D[2] \\
\text{sage: del } L[5] \\
\text{sage: for v in sorted(D.values()):} \\
\hspace{1em} \text{print}(v) \\
\hspace{1em} <0> \\
\hspace{1em} <1> \\
\hspace{1em} <3> \\
\hspace{1em} <4> \\
\hspace{1em} <6> \\
\hspace{1em} <7> \\
\hspace{1em} <8> \\
\hspace{1em} <9>
\]

\[
\text{>>> from sage.all import } * \\
\text{>>> del } D[Integer(2)] \\
\text{>>> del } L[Integer(5)] \\
\text{>>> for v in sorted(D.values()):} \\
\hspace{1em} \text{print}(v) \\
\hspace{1em} <0> \\
\hspace{1em} <1> \\
\hspace{1em} <3> \\
\hspace{1em} <4> \\
\hspace{1em} <6> \\
\hspace{1em} <7> \\
\hspace{1em} <8> \\
\hspace{1em} <9>
\]

\text{values_list()} \]

Return the list of values.

EXAMPLES:

\[
\text{sage: import sage.misc.weak_dict} \\
\text{sage: class Vals:} \\
\hspace{1em} \text{def } \_\_\_\_init\_\_\_(self, n): \\
\hspace{2em} \text{self.n = n} \\
\hspace{1em} \text{def } \_\_\_\_repr\_\_\_(self): \\
\hspace{2em} \text{return } "<\%s\> " % self.n \\
\hspace{1em} \text{def } \_\_\_\_<\_\_\_\_(self, other): \\
\hspace{2em} \text{return } \text{self.n} < \text{other.n} \\
\hspace{1em} \text{def } \_\_\_\_eq\_\_\_(self, other): \\
\hspace{2em} \text{return } \text{self.n} == \text{other.n} \\
\hspace{1em} \text{def } \_\_\_\_ne\_\_\_\_(self, other): \\
\hspace{2em} \text{return } \text{self.val()} != \text{other.val()} \\
\text{sage: L = [Vals(n) for n in range(10)]} \\
\text{sage: D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))} \\
\]

\[
\text{>>> from sage.all import } * \\
\text{>>> import sage.misc.weak_dict} \\
\]

(continues on next page)
2.9 Fast Expression Evaluation

2.9.1 Fast Expression Evaluation

For many applications such as numerical integration, differential equation approximation, plotting a 3d surface, optimization problems, Monte-Carlo simulations, etc., one wishes to pass around and evaluate a single algebraic expression many, many times at various floating point values. Other applications may need to evaluate an expression many times in interval arithmetic, or in a finite field. Doing this via recursive calls over a python representation of the object (even if Maxima or other outside packages are not involved) is extremely inefficient.

This module provides a function, \texttt{fast_callable()}, to transform such expressions into a form where they can be evaluated quickly:

\begin{verbatim}
# needs sage.symbolic
f = sin(x) + 3*x**2
ff = fast_callable(f, vars=[x])
ff(3.5)
36.3992167723104
ff(RIF(3.5))
36.39921677231038?
\end{verbatim}

\begin{verbatim}
from sage.all import *
# needs sage.symbolic
f = sin(x) + Integer(3)*x**Integer(2)
\end{verbatim}
By default, `fast_callable()` only removes some interpretive overhead from the evaluation, but all of the individual arithmetic operations are done using standard Sage arithmetic. This is still a huge win over `sage.calculus`, which evidently has a lot of overhead. Compare the cost of evaluating Wilkinson’s polynomial (in unexpanded form) at $x = 30$:

```python
sage: # needs sage.symbolic
sage: wilk = prod((x-i) for i in [1 .. 20]); wilk
(x - 1)*(x - 2)*(x - 3)*(x - 4)*(x - 5)*(x - 6)*(x - 7)*(x - 8)*(x - 9)*(x - 10)*(x - 11)*(x - 12)*(x - 13)*(x - 14)*(x - 15)*(x - 16)*(x - 17)*(x - 18)*(x - 19)*(x - 20)
sage: timeit(wilk.subs(x=30))  # random # long time
625 loops, best of 3: 1.43 ms per loop
sage: fc_wilk = fast_callable(wilk, vars=[x])
sage: timeit('fc_wilk(30)')  # random # long time
625 loops, best of 3: 9.72 us per loop
```

You can specify a particular domain for the evaluation using `domain=`:

```python
sage: fc_wilk_zz = fast_callable(wilk, vars=[x], domain=ZZ)  #...
```

The meaning of `domain=D` is that each intermediate and final result is converted to type `D`. For instance, the previous example of $\sin(x) + 3 \times x^2$ with `domain=D` would be equivalent to $D(D(\sin(D(x)))) + D(D(3) \times D(D(x)^2)))$. (This example also demonstrates the one exception to the general rule: if an exponent is an integral constant, then it is not wrapped with $D()$.)

At first glance, this seems like a very bad idea if you want to compute quickly. And it is a bad idea, for types where we don’t have a special interpreter. It’s not too bad of a slowdown, though. To mitigate the costs, we check whether the value already has the correct parent before we call `D`.

We don’t yet have a special interpreter with domain `ZZ`, so we can see how that compares to the generic `fc_wilk` example above:

```python
sage: timeit('fc_wilk_zz(30)')  # random # long time  #...
625 loops, best of 3: 15.4 us per loop
```

2.9. Fast Expression Evaluation 323
However, for other types, using `domain=D` will get a large speedup, because we have special-purpose interpreters for those types. One example is RDF. Since with `domain=RDF` we know that every single operation will be floating-point, we can just execute the floating-point operations directly and skip all the Python object creations that you would get from actually using RDF objects:

```
sage: fc_wilk_rdf = fast_callable(wilk, vars=[x], domain=RDF)  # needs sage.symbolic
sage: timeit('fc_wilk_rdf(30.0)')  # random # long time  # needs sage.symbolic
625 loops, best of 3: 7 us per loop
```

The domain does not need to be a Sage type; for instance, `domain=float` also works. (We actually use the same fast interpreter for `domain=float` and `domain=RDF`; the only difference is that when `domain=RDF` is used, the return value is an RDF element, and when `domain=float` is used, the return value is a Python float.)

```
sage: fc_wilk_float = fast_callable(wilk, vars=[x], domain=float)  # needs sage.symbolic
sage: timeit('fc_wilk_float(30.0)')  # random # long time  # needs sage.symbolic
625 loops, best of 3: 5.04 us per loop
```

We also have support for RR:

```
sage: fc_wilk_rr = fast_callable(wilk, vars=[x], domain=RR)  # needs sage.symbolic
sage: timeit('fc_wilk_rr(30.0)')  # random # long time  # needs sage.symbolic
625 loops, best of 3: 13 us per loop
```

For CC:
Currently, `fast_callable()` can accept two kinds of objects: polynomials (univariate and multivariate) and symbolic expressions (elements of the Symbolic Ring). For polynomials, you can omit the `vars` argument; the variables will default to the ring generators (in the order used when creating the ring).

```python
sage: K.<x,y,z> = QQ[]
sage: p = 10*y + 100*z + x
sage: fp = fast_callable(p)
sage: fp(1,2,3)
321
```

But you can also specify the variable names to override the default ordering (you can include extra variable names here, too).

```python
sage: fp = fast_callable(p, vars=('x','y','z'))
```

For symbolic expressions, you need to specify the variable names, so that `fast_callable()` knows what order to use.

```python
sage: fc_wilk_cdf = fast_callable(wilk, vars=[x], domain=CDF)  # needs sage.symbolic
sage: timeit('fc_wilk_cdf(30.0)')  # random # long time  # needs sage.symbolic
625 loops, best of 3: 10.2 us per loop
```

And support for CDF:

```python
sage: fc_wilk_cdf = fast_callable(wilk, vars=[x], domain=CDF)  # needs sage.symbolic
sage: timeit('fc_wilk_cdf(30.0)')  # random # long time  # needs sage.symbolic
625 loops, best of 3: 10.2 us per loop
```

2.9. Fast Expression Evaluation
Utilities, Release 10.4

```python
sage: # needs sage.symbolic
sage: var('y,z,x')
(y, z, x)
sage: f = 10*y + 100*z + x
sage: ff = fast_callable(f, vars=(x,y,z))
321
sage: ff(1,2,3)
321
```

```python
>>> from sage.all import *
>>> # needs sage.symbolic
>>> var('y,z,x')
(y, z, x)
>>> f = Integer(10)*y + Integer(100)*z + x
>>> ff = fast_callable(f, vars=(x,y,z))
>>> ff(Integer(1),Integer(2),Integer(3))
321
```

You can also specify extra variable names:

```python
sage: ff = fast_callable(f, vars=('x','w','z','y'))
# needs sage.symbolic
sage: ff(1,2,3,4)
341
```

This should be enough for normal use of `fast_callable()`; let's discuss some more advanced topics.

Sometimes it may be useful to create a fast version of an expression without going through symbolic expressions or polynomials; perhaps because you want to describe to `fast_callable()` an expression with common subexpressions.

Internally, `fast_callable()` works in two stages: it constructs an expression tree from its argument, and then it builds a fast evaluator from that expression tree. You can bypass the first phase by building your own expression tree and passing that directly to `fast_callable()`, using an `ExpressionTreeBuilder`.

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,y,z))
```

An `ExpressionTreeBuilder` has three interesting methods: `constant()`, `var()`, and `call()`. All of these methods return `ExpressionTree` objects.

The `var()` method takes a string, and returns an expression tree for the corresponding variable.

```python
sage: x = etb.var('x')
sage: y = etb.var('y')
sage: z = etb.var('y')
```
Expression trees support Python’s numeric operators, so you can easily build expression trees representing arithmetic expressions.

```python
>>> from sage.all import *
>>> x = etb.var('x')
>>> y = etb.var('y')
>>> z = etb.var('y')
```

The constant() method takes a Sage value, and returns an expression tree representing that value.

```python
sage: v2 = etb.constant(3.14159) * x + etb.constant(1729) * y
```

The call() method takes a sage/Python function and zero or more expression trees, and returns an expression tree representing the function call.

```python
sage: v3 = etb.call(sin, v1+v2)
sage: v3  # needs sage.symbolic
sin(add(add(mul(add(v_0, v_1), add(v_1, v_1)), floordiv(v_1, v_1)),
       add(mul(3.14159000000000, v_0), mul(1729, v_1))))
```

Many sage/Python built-in functions are specially handled; for instance, when evaluating an expression involving `sin()` over RDF, the C math library function `sin()` is called. Arbitrary functions are allowed, but will be much slower since they will call back to Python code on every call; for example, the following will work.

```python
sage: def my_sqrt(x): return pow(x, 0.5)
sage: e = etb.call(my_sqrt, v1); e
{my_sqrt}(add(mul(add(v_0, v_1), add(v_1, v_1)), floordiv(v_1, v_1)),
        add(mul(3.14159000000000, v_0), mul(1729, v_1))))
```

To provide `fast_callable()` for your own class (so that `fast_callable(x)` works when `x` is an instance of your class), implement a method `_fast_callable_(self, etb)` for your class. This method takes an ExpressionTreeBuilder, and returns an expression tree built up using the methods described above.
EXAMPLES:

```python
sage: var('x') # needs sage.symbolic
x
sage: f = fast_callable(sqrt(x^7+1), vars=[x], domain=float) # needs sage.symbolic

>>> from sage.all import *

>>> var('x') # needs sage.symbolic
x

>>> f = fast_callable(sqrt(x**Integer(7)+Integer(1)), vars=[x], domain=float) # needs sage.symbolic

sage: f(1) # needs sage.symbolic
1.4142135623730951

sage: f.op_list() # needs sage.symbolic
[('load_arg', 0), ('ipow', 7), ('load_const', 1.0), 'add', 'sqrt', 'return']
```

To interpret that last line, we load argument 0 (\(x\) in this case) onto the stack, push the constant 7.0 onto the stack, call the `pow()` function (which takes 2 arguments from the stack), push the constant 1.0, add the top two arguments of the stack, and then call `sqrt()`.

Here we take \(\sin()\) of the first argument and add it to \(f\):

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
e_tb = ExpressionTreeBuilder('x')
sage: x = e_tb.var('x')
sage: f = e_tb.call(sqrt, x^7 + 1)
sage: g = e_tb.call(sin, x)
sage: fast_callable(f+g).op_list()

[('load_arg', 0), ('ipow', 7), ('load_const', 1.0), 'add', ('py_call', <function sqrt at ...>, 1), ('load_arg', 0), ('py_call', sin, 1), 'add', 'return']
```

AUTHOR:
Todo: The following bits of text were written for the module docstring. They are not true yet, but I hope they will be true someday, at which point I will move them into the main text.

The final interesting method of *ExpressionTreeBuilder* is *choice()*. This produces conditional expressions, like the C \( \text{COND} ? \ T : \ F \) expression or the Python \( T \text{ if COND else F} \). This lets you define piecewise functions using *fast_callable()*.

```python
sage: v4 = etb.choice(v3 >= etb.constant(0), v1, v2) # not tested
>>> from sage.all import *
>>> v4 = etb.choice(v3 >= etb.constant(Integer(0)), v1, v2) # not tested
```

The arguments are \((\text{COND}, \ T, \ F)\) (the same order as in C), so the above means that if \(v3\) evaluates to a nonnegative number, then \(v4\) will evaluate to the result of \(v1\); otherwise, \(v4\) will evaluate to the result of \(v2\).

Let's see an example where we see that *fast_callable()* does not evaluate common subexpressions more than once. We'll make a *fast_callable()* expression that gives the result of 16 iterations of the Mandelbrot function.

```python
sage: etb = ExpressionTreeBuilder('c')
sage: z = etb.constant(0)
sage: c = etb.var('c')
sage: for i in range(16):
    ...:     z = z*z + c
sage: mand = fast_callable(z, domain=CDF)  # needs sage.rings.complex_double
>>> from sage.all import *
>>> etb = ExpressionTreeBuilder('c')
>>> z = etb.constant(Integer(0))
>>> c = etb.var('c')
>>> for i in range(Integer(16)):
    ...:     z = z*z + c
>>> mand = fast_callable(z, domain=CDF)  # needs...

\(--\) sage.rings.complex_double
```

Now \(f\) does 32 complex arithmetic operations on each call (16 additions and 16 multiplications). However, if \(z*z\) produced code that evaluated \(z\) twice, then this would do many thousands of arithmetic operations instead.

Note that the handling for common subexpressions only checks whether expression trees are the same Python object; for instance, the following code will evaluate \(x+1\) twice:

```python
sage: etb = ExpressionTreeBuilder('x')
sage: x = etb.var('x')
sage: (x+1)*(x+1)
mul(add(v_0, 1), add(v_0, 1))

>>> from sage.all import *
>>> etb = ExpressionTreeBuilder('x')
>>> x = etb.var('x')
>>> (x+Integer(1))*(x+Integer(1))
mul(add(v_0, 1), add(v_0, 1))
```

but this code will only evaluate \(x+1\) once:
Utilities, Release 10.4

```python
sage: v = x+1; v*v
mul(add(v_0, 1), add(v_0, 1))
```

```python
>>> from sage.all import *

>>> v = x+Integer(1); v*v
mul(add(v_0, 1), add(v_0, 1))
```

```python
class sage.ext.fast_callable.CompilerInstrSpec(n_inputs, n_outputs, parameters)

Bases: object

Describe a single instruction to the fast_callable() code generator.

An instruction has a number of stack inputs, a number of stack outputs, and a parameter list describing extra arguments that must be passed to the InstructionStream.instr() method (that end up as extra words in the code).

The parameter list is a list of strings. Each string is one of the following:

- 'args' – The instruction argument refers to an input argument of the wrapper class; it is just appended to the code.
- 'constants', 'py_constants' – The instruction argument is a value; the value is added to the corresponding list (if it’s not already there) and the index is appended to the code.
- 'n_inputs', 'n_outputs' – The instruction actually takes a variable number of inputs or outputs (the n_inputs and n_outputs attributes of this instruction are ignored). The instruction argument specifies the number of inputs or outputs (respectively); it is just appended to the code.

class sage.ext.fast_callable.Expression

Bases: object

Represent an expression for fast_callable().

Supports the standard Python arithmetic operators; if arithmetic is attempted between an Expression and a non-Expression, the non-Expression is converted to an expression (using the __call__() method of the Expression's ExpressionTreeBuilder).

EXAMPLES:

```python
sage: # needs sage.symbolic
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: x = etb.var(x)

sage: etb(x)

v_0

sage: etb(3)

3

sage: etb.call(sin, x)

sin(v_0)

sage: (x+1)/(x-1)

div(add(v_0, 1), sub(v_0, 1))

sage: x//5

floordiv(v_0, 5)

sage: -abs(~x)

neg(abs(inv(v_0)))
```

```python
>>> from sage.all import *

>>> # needs sage.symbolic

(continues on next page)```
```python
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=(x,))
>>> x = etb.var(x)
>>> etb(x)
\n>>> etb(Integer(3))
3
>>> etb.call(sin, x)
sin(v_0)
>>> (x+Integer(1))/(x-Integer(1))
div(add(v_0, 1), sub(v_0, 1))
>>> x//Integer(5)
floordiv(v_0, 5)
>>> -abs(~x)
neg(abs(inv(v_0)))
```

**abs()**

Compute the absolute value of an Expression.

**EXAMPLES:**

```python
sage: # needs sage.symbolic
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: x = etb(x)
sage: abs(x)
abs(v_0)
sage: x.abs()
abs(v_0)
sage: x.__abs__()
abs(v_0)
```

```python
>>> from sage.all import *

>>> from sage.ext.fast_callable import ExpressionTreeBuilder

>>> etb = ExpressionTreeBuilder(vars=(x,))

>>> abs(x)
abs(v_0)

>>> x.abs()
abs(v_0)

>>> x.__abs__()
abs(v_0)
```

**class** `sage.ext.fast_callable.ExpressionCall`

**Bases:** `Expression`

An Expression that represents a function call.

**EXAMPLES:**

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))

# needs sage.symbolic

sage: type(etb.call(sin, x))
<class 'sage.ext.fast_callable.ExpressionCall'>
```

2.9. Fast Expression Evaluation
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=(x,))  # needs sage.symbolic
>>> type(etb.call(sin, x))  # needs sage.symbolic
<class 'sage.ext.fast_callable.ExpressionCall'>

arguments()
Return the arguments from this ExpressionCall.

EXAMPLES:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))  # needs sage.symbolic
sage: etb.call(sin, x).arguments()  # needs sage.symbolic
[v_0]
```

function()
Return the function from this ExpressionCall.

EXAMPLES:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))  # needs sage.symbolic
sage: etb.call(sin, x).function()  # needs sage.symbolic
sin
```

class sage.ext.fast_callable.ExpressionChoice
Bases: Expression
A conditional expression.
(It's possible to create choice nodes, but they don't work yet.)

EXAMPLES:
Utilities, Release 10.4

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))  # needs sage.symbolic
sage: etb.choice(etb.call(operator.eq, x, 0), 0, 1/x)  # needs sage.symbolic
(0 if (eq(v_0, 0) else div(1, v_0))
```

```python
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=(x,))  # needs sage.symbolic
>>> etb.choice(etb.call(operator.eq, x, Integer(0)), Integer(0), Integer(1)/x)  # needs sage.symbolic
(0 if (eq(v_0, 0) else div(1, v_0))
```

### condition() ▼

Return the condition of an `ExpressionChoice`.

**EXAMPLES:**

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))  # needs sage.symbolic
sage: x = etb(x)  # needs sage.symbolic
sage: etb.choice(x, ~x, 0).condition()  # needs sage.symbolic
v_0
```

```python
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=(x,))  # needs sage.symbolic
>>> etb.choice(x, ~x, Integer(0)).condition()  # needs sage.symbolic
v_0
```

### if_false() ▼

Return the false branch of an `ExpressionChoice`.

**EXAMPLES:**

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))  # needs sage.symbolic
sage: x = etb(x)  # needs sage.symbolic
sage: etb.choice(x, ~x, 0).if_false()  # needs sage.symbolic
0
```

```python
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=(x,))  # needs sage.symbolic
>>> etb.choice(x, ~x, 0).if_false()  # needs sage.symbolic
0
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

```python
>>> x = etb(x)
# needs sage.symbolic
>>> etb.choice(x, ~x, Integer(0)).if_false()
-> # needs sage.symbolic
0
```

if_true()

Return the true branch of an `ExpressionChoice`.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
# needs sage.symbolic
sage: x = etb(x)
# needs sage.symbolic
sage: etb.choice(x, ~x, 0).if_true()  # needs sage.symbolic
inv(v_0)
```

```python
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>>
```

class sage.ext.fast_callable.ExpressionConstant

Bases: `Expression`

An Expression that represents an arbitrary constant.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))  # needs sage.symbolic
sage: type(etb(3))  # needs sage.symbolic
<class 'sage.ext.fast_callable.ExpressionConstant'>
```

```python
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>>
```

value()

Return the constant value of an `ExpressionConstant`.

EXAMPLES:
```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))  # needs sage.symbolic
sage: etb(3).value()  # needs sage.symbolic
3

```
exponent()

Return the exponent from this ExpressionIPow.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))  # needs sage.symbolic
sage: (etb(x)^(-1)).exponent()  # needs sage.symbolic
-1
```

```python
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(x)
>>> x = etb.var('x')
>>> (x+Integer(3))*Integer(5)
mul(add(v_0, 3), 5)
```

class sage.ext.fast_callable.ExpressionTreeBuilder

Bases: object

A class with helper methods for building Expressions.

An instance of this class is passed to _fast_callable_() methods; you can also instantiate it yourself to create your own expressions for fast_callable(), bypassing _fast_callable_().

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(x)
sage: (x+3)*5
mul(add(v_0, 3), 5)
```

```python
call(fn, *args)

Construct a call node, given a function and a list of arguments.

The arguments will be converted to Expressions using ExpressionTreeBuilder.__call__().

As a special case, notice if the function is operator.pow() and the second argument is integral, and construct an ExpressionIPow instead.

EXAMPLES:

```python
sage: # needs sage.symbolic
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: etb.call(cos, x)
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

```python
sage: etb.call(sin, 1)
sin(1)
sage: etb.call(sin, etb(1))
sin(1)
sage: etb.call(factorial, x+57)
{factorial}(add(v_0, 57))
sage: etb.call(operator.pow, x, 543)
ipow(v_0, 543)
```

```python
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=(x,))
>>> etb.choice(etb.call(operator.eq, x, 0), 0, 1/x)
```

`choice` *(cond, iftrue, iffalse)*

Construct a choice node (a conditional expression), given the condition, and the values for the true and false cases.

(It’s possible to create choice nodes, but they don’t work yet.)

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb.choice(etb.call(operator.eq, x, Integer(0)), Integer(0), Integer(1)/x)
```

`constant` *(c)*

Turn the argument into an `ExpressionConstant`, converting it to our domain if we have one.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder('x')
sage: etb.constant(pi)
```

(continues on next page)
pi
sage: etb = ExpressionTreeBuilder('x', domain=RealField(200))  # needs sage.rings.real_mpfr
sage: etb.constant(pi)  # needs sage.rings.real_mpfr sage.symbolic
3.1415926535897932384626433832795028841971693993751058209749

>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder('x')
>>> etb.constant(pi)  # needs sage.symbolic
pi
>>> etb = ExpressionTreeBuilder('x', domain=RealField(Integer(200)))  # needs sage.rings.real_mpfr
>>> etb.constant(pi)  # needs sage.rings.real_mpfr sage.symbolic
3.1415926535897932384626433832795028841971693993751058209749

var(v)

Turn the argument into an ExpressionVariable. Look it up in the list of variables. (Variables are matched by name.)

EXAMPLES:

sage: # needs sage.symbolic
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: var('a,b,some_really_long_name')
(a, b, some_really_long_name)

sage: x = polygen(QQ)
sage: etb = ExpressionTreeBuilder(vars=('a', 'b', 'some_really_long_name', x))

v_2
sage: etb.var('some_really_long_name')
        
v_2
sage: etb.var(x)
        
v_3
sage: etb.var('y')
Traceback (most recent call last):
... ValueError: Variable 'y' not found...

(continues on next page)
Traceback (most recent call last):
...
ValueError: Variable 'y' not found...

class sage.ext.fast_callable.ExpressionVariable

Bases: Expression

An Expression that represents a variable.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))  #...
˓→needs sage.symbolic
sage: type(etb.var(x))  #...
˓→needs sage.symbolic
<class 'sage.ext.fast_callable.ExpressionVariable'>
```

```
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=(x,))  #...
˓→needs sage.symbolic
>>> type(etb.var(x))  #...
˓→needs sage.symbolic
<class 'sage.ext.fast_callable.ExpressionVariable'>
```

variable_index()

Return the variable index of an ExpressionVariable.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))  #...
˓→needs sage.symbolic
sage: etb(x).variable_index()  #...
˓→needs sage.symbolic
0
```

```
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=(x,))  #...
˓→needs sage.symbolic
>>> etb(x).variable_index()  #...
˓→needs sage.symbolic
0
```

class sage.ext.fast_callable.FastCallableFloatWrapper(ff, imag_tol)

Bases: object

A class to alter the return types of the fast-callable functions.

When applying numerical routines (including plotting) to symbolic expressions and functions, we generally first convert them to a faster form with `fast_callable()`. That function takes a domain parameter that forces the end (and all intermediate) results of evaluation to a specific type. Though usually always want the end result to be of type float, correctly choosing the domain presents some problems:
• float is a bad choice because it’s common for real functions to have complex terms in them. Moreover precision issues can produce terms like $1.0 + 1e-12*I$ that are hard to avoid if calling real() on everything is infeasible.

• complex has essentially the same problem as float. There are several symbolic functions like min_symbolic(), max_symbolic(), and floor() that are unable to operate on complex numbers.

• None leaves the types of the inputs/outputs alone, but due to the lack of a specialized interpreter, slows down evaluation by an unacceptable amount.

• CDF has none of the other issues, because CDF has its own specialized interpreter, a lexicographic ordering (for min/max), and supports floor(). However, most numerical functions cannot handle complex numbers, so using CDF would require us to wrap every evaluation in a CDF-to-float conversion routine. That would slow things down less than a domain of None would, but is unattractive mainly because of how invasive it would be to “fix” the output everywhere.

Creating a new fast-callable interpreter that has different input and output types solves most of the problems with a CDF domain, but fast_callable() and the interpreter classes in sage.ext.interpreters are not really written with that in mind. The domain parameter to fast_callable(), for example, is expecting a single Sage ring that corresponds to one interpreter. You can make it accept, for example, a string like “CDF-to-float”, but the hacks required to make that work feel wrong.

Thus we arrive at this solution: a class to wrap the result of fast_callable(). Whenever we need to support intermediate complex terms in a numerical routine, we can set domain=CDF while creating its fast-callable incarnation, and then wrap the result in this class. The __call__() method of this class then ensures that the CDF output is converted to a float if its imaginary part is within an acceptable tolerance.

**EXAMPLES:**

An error is thrown if the answer is complex:

```
sage: # needs sage.symbolic
sage: from sage.ext.fast_callable import FastCallableFloatWrapper
sage: f = sqrt(x)
sage: ff = fast_callable(f, vars=[x], domain=CDF)
sage: fff = FastCallableFloatWrapper(ff, imag_tol=1e-8)
sage: fff(1)
1.0
sage: fff(-1)
Traceback (most recent call last):
... ValueError: complex fast-callable function result 1.0*I for arguments (-1,)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> from sage.ext.fast_callable import FastCallableFloatWrapper
>>> f = sqrt(x)
>>> ff = fast_callable(f, vars=[x], domain=CDF)
>>> fff = FastCallableFloatWrapper(ff, imag_tol=RealNumber('1e-8'))
>>> fff(Integer(1))
1.0
>>> fff(-Integer(1))
Traceback (most recent call last):
... ValueError: complex fast-callable function result 1.0*I for arguments (-1,)
```

class sage.ext.fast_callable.InstructionStream
Utilities, Release 10.4

Bases: object

An InstructionStream takes a sequence of instructions (passed in by a series of method calls) and computes the data structures needed by the interpreter. This is the stage where we switch from operating on Expression trees to a linear representation. If we had a peephole optimizer (we don’t) it would go here.

Currently, this class is not very general; it only works for interpreters with a fixed set of memory chunks (with fixed names). Basically, it only works for stack-based expression interpreters. It should be generalized, so that the interpreter metadata includes a description of the memory chunks involved and the instruction stream can handle any interpreter.

Once you’re done adding instructions, you call get_current() to retrieve the information needed by the interpreter (as a Python dictionary).

current_op_list()

Return the list of instructions that have been added to this InstructionStream so far.

It’s OK to call this, then add more instructions.

EXAMPLES:

```python
sage: from sage.ext.interpreters.wrapper_el import metadata
sage: from sage.ext.interpreters.wrapper_rdf import metadata
sage: instr_stream = InstructionStream(metadata, 1)
sage: instr_stream.instr(load_arg, 0)
sage: instr_stream.instr(py_call, math.sin, 1)
sage: instr_stream.instr(abs)
sage: instr_stream.instr(return)
```

```python
[(load_arg, 0), (py_call, <built-in function sin>, 1), 'abs', 'return']
```

get_current()

Return the current state of the InstructionStream, as a dictionary suitable for passing to a wrapper class.

NOTE: The dictionary includes internal data structures of the InstructionStream; you must not modify it.

EXAMPLES:

```python
sage: from sage.ext.interpreters.wrapper_el import metadata
sage: from sage.ext.interpreters.wrapper_rdf import metadata
sage: instr_stream = InstructionStream(metadata, 1)
sage: instr_stream.instr(load_arg, 0)
sage: instr_stream.instr(py_call, math.sin, 1)
```

```python
[(load_arg, 0), (py_call, <built-in function sin>, 1), 'abs', 'return']
```
```python
sage: from sage.ext.fast_callable import InstructionStream
sage: instr_stream = InstructionStream(metadata, 1)

sage: instr_stream.get_current()
{'args': 1,
 'code': [],
 'constants': [],
 'domain': None,
 'py_constants': [],
 'stack': 0}

sage: instr_stream.instr('load_arg', 0)

sage: instr_stream.instr('py_call', math.sin, 1)

sage: instr_stream.instr('abs')

sage: instr_stream.instr('return')

sage: instr_stream.current_op_list()
[('load_arg', 0), ('py_call', <built-in function sin>, 1), 'abs', 'return']

sage: instr_stream.get_current()
{'args': 1,
 'code': [0, 0, 3, 0, 1, 12, 2],
 'constants': [],
 'domain': None,
 'py_constants': [<built-in function sin>],
 'stack': 1}
```

```python
>>> from sage.all import *

>>> from sage.ext.interpreters.wrapper_el import metadata

>>> from sage.ext.interpreters.wrapper_rdf import metadata

get_metadata()

Return the interpreter metadata being used by the current InstructionStream.

The code generator sometimes uses this to decide which code to generate.

EXAMPLES:
```
sage: from sage.ext.interpreters.wrapper_el import metadata
sage: from sage.ext.interpreters.wrapper_rdf import metadata
  needs sage.modules
sage: from sage.ext.fast_callable import InstructionStream
sage: instr_stream = InstructionStream(metadata, 1)
sage: md = instr_stream.get_metadata()
sage: type(md)
<class 'sage.ext.fast_callable.InterpreterMetadata'>

>>> from sage.all import *
>>> from sage.ext.interpreters.wrapper_el import metadata
>>> from sage.ext.interpreters.wrapper_rdf import metadata
  needs sage.modules
>>> from sage.ext.fast_callable import InstructionStream
>>> instr_stream = InstructionStream(metadata, Integer(1))
>>> md = instr_stream.get_metadata()
>>> type(md)
<class 'sage.ext.fast_callable.InterpreterMetadata'>

has_instr(opname)

Check whether this InstructionStream knows how to generate code for a given instruction.

EXAMPLES:

sage: from sage.ext.interpreters.wrapper_el import metadata
sage: from sage.ext.interpreters.wrapper_rdf import metadata
  needs sage.modules
sage: from sage.ext.fast_callable import InstructionStream
sage: instr_stream = InstructionStream(metadata, 1)
sage: instr_stream.has_instr(return)
True
sage: instr_stream.has_instr(factorial)
False
sage: instr_stream.has_instr(abs)
True

>>> from sage.all import *
>>> from sage.ext.interpreters.wrapper_el import metadata
>>> from sage.ext.interpreters.wrapper_rdf import metadata
  needs sage.modules
>>> from sage.ext.fast_callable import InstructionStream
>>> instr_stream = InstructionStream(metadata, Integer(1))
>>> instr_stream.has_instr(return)
True
>>> instr_stream.has_instr(factorial)
False
>>> instr_stream.has_instr(abs)
True

instr(opname, *args)

Generate code in this InstructionStream for the given instruction and arguments.

The opname is used to look up a CompilerInstrSpec; the CompilerInstrSpec describes how to interpret the arguments. (This is documented in the class docstring for CompilerInstrSpec.)

EXAMPLES:
Utilities, Release 10.4

```python
sage: from sage.ext.interpreters.wrapper_el import metadata
sage: from sage.ext.interpreters.wrapper_rdf import metadata
→ needs sage.modules
sage: from sage.ext.fast_callable import InstructionStream
sage: instr_stream = InstructionStream(metadata, 1)
sage: instr_stream.instr('load_arg', 0)
sage: instr_stream.instr('sin')
→ needs sage.symbolic
sage: instr_stream.instr('py_call', math.sin, 1)
sage: instr_stream.instr('abs')
sage: instr_stream.instr('factorial')
Traceback (most recent call last):
...
KeyError: 'factorial'
sage: instr_stream.instr('return')
sage: instr_stream.current_op_list()
→ needs sage.symbolic
[('load_arg', 0), 'sin', ('py_call', <built-in function sin>, 1), 'abs', 'return']

>>> from sage.all import *
>>> from sage.ext.interpreters.wrapper_el import metadata
>>> from sage.ext.interpreters.wrapper_rdf import metadata
→ needs sage.modules
>>> from sage.ext.fast_callable import InstructionStream
>>> instr_stream = InstructionStream(metadata, Integer(1))
>>> instr_stream.load_arg(Integer(5))
>>> instr_stream.current_op_list()
[('load_arg', 5)]
>>> instr_stream.load_arg(Integer(3))
>>> instr_stream.current_op_list()
[('load_arg', 5), ('load_arg', 3)]
```

`load_arg(n)`

Add a 'load_arg' instruction to this `InstructionStream`.

EXAMPLES:
>>> from sage.all import *
>>> from sage.ext.interpreters.wrapper_el import metadata
>>> from sage.ext.interpreters.wrapper_rdf import metadata #
needs sage.modules
>>> from sage.ext.fast_callable import InstructionStream
>>> instr_stream = InstructionStream(metadata, Integer(12))
>>> instr_stream.load_arg(Integer(5))
>>> instr_stream.current_op_list()
[('load_arg', 5)]
>>> instr_stream.load_arg(Integer(3))
>>> instr_stream.current_op_list()
[('load_arg', 5), ('load_arg', 3)]

load_const(c)
Add a 'load_const' instruction to this InstructionStream.

EXAMPLES:

sage: from sage.ext.interpreters.wrapper_el import metadata
sage: from sage.ext.interpreters.wrapper_rdf import metadata #
needs sage.modules
sage: from sage.ext.fast_callable import InstructionStream, op_list
sage: instr_stream = InstructionStream(metadata, 1)
sage: instr_stream.load_const(5)
sage: instr_stream.current_op_list()
[('load_const', 5)]
sage: instr_stream.load_const(7)
sage: instr_stream.load_const(5)
sage: instr_stream.current_op_list()
[('load_const', 5), ('load_const', 7), ('load_const', 5)]

>>> from sage.all import *
>>> from sage.ext.interpreters.wrapper_el import metadata
>>> from sage.ext.interpreters.wrapper_rdf import metadata #
needs sage.modules
>>> from sage.ext.fast_callable import InstructionStream, op_list
>>> instr_stream = InstructionStream(metadata, Integer(1))
>>> instr_stream.load_const(Integer(5))
>>> instr_stream.current_op_list()
[('load_const', 5)]
>>> instr_stream.load_const(Integer(7))
>>> instr_stream.load_const(Integer(5))
>>> instr_stream.current_op_list()
[('load_const', 5), ('load_const', 7), ('load_const', 5)]

Note that constants are shared: even though we load 5 twice, it only appears once in the constant table.

sage: instr_stream.get_current()[['constants']]
[5, 7]

>>> from sage.all import *
>>> instr_stream.get_current()[['constants']]
[5, 7]

class sage.ext.fast_callable.IntegerPowerFunction(n)
Bases: object
This class represents the function $x^n$ for an arbitrary integral power $n$.

That is, `IntegerPowerFunction(2)` is the squaring function; `IntegerPowerFunction(-1)` is the reciprocal function.

**EXAMPLES:**

```python
sage: from sage.ext.fast_callable import IntegerPowerFunction
sage: square = IntegerPowerFunction(2)
sage: square
(\^2)
sage: square(pi)  # needs sage.symbolic
\pi^2
sage: square(1)  # needs sage.symbolic
1
sage: square(RIF(-1, 1)).str(style='brackets')  # needs sage.rings.real_interval_field
'[-1.0000000000000000 .. 1.0000000000000000]'
sage: IntegerPowerFunction(-1)
(^{-1})
sage: IntegerPowerFunction(-1)(22/7)
7/22
sage: v = Integers(123456789)(54321)
sage: v^9876543210
79745229
sage: IntegerPowerFunction(9876543210)(v)
79745229
```

```python
>>> from sage.all import *
>>> from sage.ext.fast_callable import IntegerPowerFunction
>>> square = IntegerPowerFunction(Integer(2))
>>> square
(\^2)
>>> square(pi)  # needs sage.symbolic
\pi^2
>>> square(1)  # needs sage.symbolic
1
>>> square(RIF(-Integer(1), Integer(1))).str(style='brackets')  # needs sage.rings.real_interval_field
'[-0.0000000000000000 .. 1.0000000000000000]'
>>> IntegerPowerFunction(-Integer(1))
(^{-1})
>>> IntegerPowerFunction(-Integer(1))(Integer(22)/Integer(7))
7/22
>>> v = Integers(Integer(123456789))(Integer(54321))
>>> v**Integer(9876543210)
79745229
>>> IntegerPowerFunction(Integer(9876543210))(v)
79745229
```

class sage.ext.fast_callable.InterpreterMetadata
    Bases: object

    The interpreter metadata for a `fast_callable()` interpreter.
Currently consists of a dictionary mapping instruction names to (CompilerInstrSpec, opcode) pairs, a list mapping opcodes to (instruction name, CompilerInstrSpec) pairs, and a range of exponents for which the 'ipow' instruction can be used. This range can be False (if the 'ipow' instruction should never be used), a pair of two integers \((a, b)\), if 'ipow' should be used for \(a \leq n \leq b\), or True, if 'ipow' should always be used. When 'ipow' cannot be used, then we fall back on calling IntegerPowerFunction.

See the class docstring for CompilerInstrSpec for more information.

NOTE: You must not modify the metadata.

by_opcode

by_opname

ipow_range

class sage.ext.fast_callable.Wrapper

Bases: object

The parent class for all fast_callable() wrappers.

Implements shared behavior (currently only debugging).

get_orig_args()

Get the original arguments used when initializing this wrapper.

(Probably only useful when writing doctests.)

EXAMPLES:

```python
sage: fast_callable(sin(x)/x, vars=[x], domain=RDF).get_orig_args()  # needs sage.symbolic
{'args': 1,
 'code': [0, 0, 16, 0, 0, 8, 2],
 'constants': [],
 'domain': Real Double Field,
 'py_constants': [],
 'stack': 2}
```

```python
>>> from sage.all import *
>>> fast_callable(sin(x)/x, vars=[x], domain=RDF).get_orig_args()  # needs sage.symbolic
{'args': 1,
 'code': [0, 0, 16, 0, 0, 8, 2],
 'constants': [],
 'domain': Real Double Field,
 'py_constants': [],
 'stack': 2}
```

op_list()

Return the list of instructions in this wrapper.

EXAMPLES:

```python
sage: fast_callable(cos(x) * x, vars=[x], domain=RDF).op_list()  # needs sage.symbolic
[('load_arg', 0), ('load_arg', 0), 'cos', 'mul', 'return']
```
Utilities, Release 10.4

```python
>>> from sage.all import *
>>> fast_callable(cos(x) * x, vars=[x], domain=RDF).op_list()  # needs sage.symbolic
[('load_arg', 0), ('load_arg', 0), 'cos', 'mul', 'return']
```

```python
def python_calls():
    List the Python functions that are called in this wrapper.

    (Python function calls are slow, so ideally this list would be empty. If it is not empty, then perhaps there is
    an optimization opportunity where a Sage developer could speed this up by adding a new instruction to the
    interpreter.)

EXAMPLES:
```
```python
sage: fast_callable(abs(sin(x)), vars=[x], domain=RDF).python_calls()  # needs sage.symbolic
[]
sage: fast_callable(abs(sin(factorial(x))), vars=[x]).python_calls()  # needs sage.symbolic
[factorial, sin]
```

```python
sage.ext.fast_callable.fast_callable(x, domain=None, vars=None, expect_one_var=False)
```

Given an expression `x`, compile it into a form that can be quickly evaluated, given values for the variables in `x`.

Currently, `x` can be an expression object, an element of `SR`, or a (univariate or multivariate) polynomial; this list
will probably be extended soon.

By default, `x` is evaluated the same way that a Python function would evaluate it – addition maps to `PyNumber_Add`, etc. However, you can specify `domain=D` where `D` is some Sage parent or Python type; in this case, all arithmetic is done in that domain. If we have a special-purpose interpreter for that parent (like `RDF` or `float`), `domain=`... will trigger the use of that interpreter.

If `vars` is `None` and `x` is a polynomial, then we will use the generators of `parent(x)` as the variables; otherwise, `vars` must be specified (unless `x` is a symbolic expression with only one variable, and `expect_one_var` is `True`, in which case we will use that variable).

EXAMPLES:
```
sage: # needs sage.symbolic
sage: var('x')
x
sage: expr = sin(x) + 3*x^2
sage: f = fast_callable(expr, vars=[x])
sage: f(2)
sin(2) + 12
sage: f(2.0)
12.9092974268257
```
We have special fast interpreters for `domain=float` and `domain=RDF`. (Actually it’s the same interpreter; only the return type varies.) Note that the float interpreter is not actually more accurate than the RDF interpreter; elements of RDF just don’t display all their digits. We have special fast interpreter for `domain=CDF`:
\[\pi^2e + \frac{1}{3}\pi^2 - \sqrt{2}\pi - \sqrt{2}e\]

```
sage: n(symbolic_result)  # needs sage.symbolic
21.8311204649396
```
Utilities, Release 10.4

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=('x', 'y'), domain=float)
```

```python
sage: x = etb.var('x')
sage: y = etb.var('y')
sage: expr = etb.call(sin, x**2 + y); expr
```

```python
sin(add(ipow(v_0, 2), v_1))
```

```python
sage: fc = fast_callable(expr, domain=float)
sage: fc(5, 7)
```

```python
0.5514266812416906
```

Check that `fast_callable()` also works for symbolic functions with evaluation functions:

```python
sage: def evalf_func(self, x, y, parent):
...     return parent(x*y) if parent is not None else x*y
sage: x, y = var(x, y)
sage: f = function(f, evalf_func=evalf_func)
sage: fc = fast_callable(f(x, y), vars=[x, y])
sage: fc(3, 4)
```

```python
f(3, 4)
```

And also when there are complex values involved:

```python
sage: def evalf_func(self, x, y, parent):
...     return parent(I*x*y) if parent is not None else I*x*y
sage: x, y = var(x, y)
sage: g = function('g', evalf_func=evalf_func)
sage: fc = fast_callable(g(x, y), vars=[x, y])
sage: fc(3, 4)
g(3, 4)
sage: fc2 = fast_callable(g(x, y), domain=complex, vars=[x, y])
sage: fc2(3, 4)
```

```python
12j
```

```python
sage: fc3 = fast_callable(g(x, y), domain=float, vars=[x, y])
sage: fc3(3, 4)
```

```python
Traceback (most recent call last):
```

(continues on next page)
...  
TypeError: unable to simplify to float approximation

>>> from sage.all import *
>>> # needs sage.symbolic
>>> def evalf_func(self, x, y, parent):
...    return parent(I*x*y) if parent is not None else I*x*y
>>> g = function('g', evalf_func=evalf_func)
>>> fc = fast_callable(g(x, y), vars=[x, y])
g(3, 4)
>>> fc2 = fast_callable(g(x, y), domain=complex, vars=[x, y])
>>> fc2(Integer(3), Integer(4))
12j
>>> fc3 = fast_callable(g(x, y), domain=float, vars=[x, y])
>>> fc3(Integer(3), Integer(4))
Traceback (most recent call last):
...  
TypeError: unable to simplify to float approximation

sage.ext.fast_callable.function_name(fn)

Given a function, return a string giving a name for the function.

For functions we recognize, we use our standard opcode name for the function (so operator.add() becomes 'add', and sage.functions.trig.sin() becomes 'sin').

For functions we don't recognize, we try to come up with a name, but the name will be wrapped in braces; this is a signal that we'll definitely use a slow Python call to call this function. (We may use a slow Python call even for functions we do recognize, if we're targeting an interpreter without an opcode for the function.)

Only used when printing Expressions.

EXAMPLES:

sage: from sage.ext.fast_callable import function_name
sage: function_name(operator.pow)
'pow'
sage: function_name(cos)  # needs sage.symbolic
'cos'
sage: function_name(factorial)
'{factorial}'

sage.ext.fast_callable.generate_code(expr, stream)

Generate code from an Expression tree; write the result into an InstructionStream.

In fast_callable(), first we create an Expression, either directly with an ExpressionTree-
Builder or with _fast_callable_() methods. Then we optimize the Expression in tree form. (Unfortunately, this step is currently missing – we do no optimizations.)

Then we linearize the Expression into a sequence of instructions, by walking the Expression and sending the corresponding stack instructions to an InstructionStream.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder, generate_code, InstructionStream

# needs sage.symbolic
sage: etb = ExpressionTreeBuilder('x')
sage: x = etb.var('x')
sage: expr = (x*pi) * (x+1)
sage: from sage.ext.interpreters.wrapper_py import metadata, Wrapper_py

sage: instr_stream = InstructionStream(metadata, 1)
sage: generate_code(expr, instr_stream)
sage: instr_stream.instr('return')
sage: v = Wrapper_py(instr_stream.get_current())
sage: type(v)
<class 'sage.ext.interpreters.wrapper_py.Wrapper_py'>
sage: v(7)
8*pi + 56
```

```python
>>> from sage.all import *
>>> from sage.ext.fast_callable import InstructionStream

# needs sage.symbolic
>>> etb = ExpressionTreeBuilder('x')
>>> x = etb.var('x')
>>> expr = (x+pi) * (x+Integer(1))
>>> from sage.ext.interpreters.wrapper_py import metadata, Wrapper_py

>>> instr_stream = InstructionStream(metadata, Integer(1))
>>> generate_code(expr, instr_stream)
>>> instr_stream.instr('return')
>>> v = Wrapper_py(instr_stream.get_current())
>>> type(v)
<class 'sage.ext.interpreters.wrapper_py.Wrapper_py'>
>>> v(Integer(7))
8*pi + 56
```

```

sage.ext.fast_callable.get_builtin_functions()
Return a dictionary from Sage and Python functions to opcode names.

The result is cached.

The dictionary is used in ExpressionCall.

EXAMPLES:

```
```
Utilities, Release 10.4

(continued from previous page)

```python
sage: builtins["sin"]  #...
˓→ needs sage.symbolic
'sin'
sage: builtins["ln"]
'log'
```

```python
>>> from sage.all import *
>>> from sage.ext.fast_callable import get_builtin_functions
>>> builtins = get_builtin_functions()  #...
˓→ needs sage.symbolic

['abs', 'acos', 'acosh', 'add', 'asin', 'asinh', 'atan', 'atanh', 'ceil',
'cos', 'cosh', 'cot', 'csc', 'div', 'exp', 'floor', 'floordiv', 'inv', 'log',
'mul', 'neg', 'pow', 'sec', 'sin', 'sinh', 'sqrt', 'sub', 'tan', 'tanh']

>>> builtins["sin"]  #...
˓→ needs sage.symbolic
'sin'

>>> builtins["ln"]
'log'
```

sage.ext.fast_callable.op_list(args, metadata)

Given a dictionary with the result of calling get_current() on an InstructionStream, and the corresponding interpreter metadata, return a list of the instructions, in a simple somewhat human-readable format.

For debugging only. (That is, it’s probably not a good idea to try to programmatically manipulate the result of this function; the expected use is just to print the returned list to the screen.)

There’s probably no reason to call this directly; if you have a wrapper object, call op_list() on it; if you have an InstructionStream object, call current_op_list() on it.

EXAMPLES:

```python
sage: from sage.ext.interpreters.wrapper_el import metadata
sage: from sage.ext.interpreters.wrapper_rdf import metadata  #...
˓→ needs sage.modules

>>> from sage.ext.fast_callable import InstructionStream, op_list

>>> instr_stream = InstructionStream(metadata, 1)
>>> instr_stream.instr('load_arg', 0)
>>> instr_stream.instr('abs')
>>> instr_stream.instr('return')
>>> instr_stream.current_op_list()

[("load_arg", 0), "abs", "return"]
```

```python
>>> from sage.all import *
>>> from sage.ext.interpreters.wrapper_el import metadata
>>> from sage.ext.interpreters.wrapper_rdf import metadata  #...
˓→ needs sage.modules

>>> instr_stream = InstructionStream(metadata, Integer(1))
>>> instr_stream.instr('load_arg', Integer(0))
>>> instr_stream.instr('abs')
>>> instr_stream.instr('return')
>>> instr_stream.current_op_list()

[("load_arg", 0), "abs", "return"]
```
2.9.2 Fast Numerical Evaluation

For many applications such as numerical integration, differential equation approximation, plotting a 3d surface, optimization problems, monte-carlo simulations, etc., one wishes to pass around and evaluate a single algebraic expression many, many times at various floating point values. Doing this via recursive calls over a python representation of the object (even if Maxima or other outside packages are not involved) is extremely inefficient.

The solution implemented in this module, by Robert Bradshaw (2008-10), has been superseded by `fast_callable()`. All that remains here is a compatible interface function `fast_float()`.

AUTHORS:


```python
sage.ext.fast_eval.fast_float(f, expect_one_var=False, *vars)
```

Try to create a function that evaluates f quickly using floating-point numbers, if possible.

On failure, returns the input unchanged.

This is an alternative interface to `sage.ext.fast_callable.fast_callable()`. Issue #32268 proposes to deprecate this function.

INPUT:

- f – an expression
- vars – the names of the arguments
- expect_one_var – don't give deprecation warning if vars is omitted, as long as expression has only one var

EXAMPLES:

```python
>>> from sage.ext.fast_eval import fast_float
>>> x, y = var('x, y')
... # needs sage.symbolic
>>> f = fast_float(sqrt(x**2+y**2), 'x', 'y')
... # needs sage.symbolic
>>> f(Integer(3), Integer(4))
... # needs sage.symbolic
5.0
```

Specifying the argument names is essential, as fast_float objects only distinguish between arguments by order.
```python
sage: # needs sage.symbolic
sage: f = fast_float(x-y, 'x', 'y')
sage: f(1,2)
-1.0
sage: f = fast_float(x-y, 'y', 'x')
sage: f(1,2)
1.0
```
3.1 Cython support functions

AUTHORS:

- William Stein (2006-01-18): initial version

`sage.misc.cython.compile_and_load(code, **kwds)`

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: from sage.misc.cython import compile_and_load
sage: module = compile_and_load("def f(int n):\n    return n*n")
sage: module.f(10)
100
```

```
>>> from sage.all import *
>>> from sage.misc.cython import compile_and_load
>>> module = compile_and_load("def f(int n):\n    return n*n")
>>> module.f(Integer(10))
100
```

`sage.misc.cython.cython(filename, verbose=0, 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.

INPUT:

- `filename` – the name of the file to be compiled. Should end with ‘.pyx’.
- `verbose` (integer, default 0) – level of verbosity. A negative value ensures complete silence.
- `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 or .cpp 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.

OUTPUT: a tuple (name, dir) where name is the name of the compiled module and dir is the directory containing the generated files.

sage.misc.cython.cython_compile(code, **kwds)

Given a block of Cython code (as a text string), this function compiles it using a C compiler, and includes it into the global namespace.

AUTHOR: William Stein, 2006-10-31

**Warning:** Only use this from Python code, not from extension code, since from extension code you would change the global scope (i.e., of the Sage interpreter). And it would be stupid, since you’re already writing Cython!

Also, never use this in the standard Sage library. Any code that uses this can only run on a system that has a C compiler installed, and we want to avoid making that assumption for casual Sage usage. Also, any code that uses this in the library would greatly slow down startup time, since currently there is no caching.

Todo: Need to create a clever caching system so code only gets compiled once.

sage.misc.cython.cython_import(filename, **kwds)

Compile a file containing Cython code, then import and return the module. Raises an ImportError if anything goes wrong.

INPUT:

• **filename** – a string; name of a file that contains Cython code

See the function sage.misc.cython.cython() for documentation for the other inputs.

OUTPUT:

• the module that contains the compiled Cython code.

sage.misc.cython.cython_import_all(filename, globals, **kwds)

Imports all non-private (i.e., not beginning with an underscore) attributes of the specified Cython module into the given context. This is similar to:

```python
from module import *
```

Raises an ImportError exception if anything goes wrong.

INPUT:
filename – a string; name of a file that contains Cython code

```python
sage.misc.cython.cython_lambda(vars, expr, verbose=0, **kwds)
```

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 Issue #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):

```python
sage: f = lambda x,y: x*x + y*y + x + y + 17*r*x + 3.2r
```

```python
>>> from sage.all import *

>>> f = lambda x,y: x*x + y*y + x + y + 17*x + 3.2
```

We make the same Lambda function, but in a compiled form.

```python
sage: g = cython_lambda(double x, double y, x*x + y*y + x + y + 17*x + 3.2)
```

```python
sage: g(2,3)
55.2
```

```python
sage: g(0,0)
3.2
```

```python
>>> from sage.all import *

>>> g = cython_lambda(double x, double y, x*x + y*y + x + y + 17*x + 3.2)

>>> g(Integer(2),Integer(3))
55.2

>>> g(Integer(0),Integer(0))
3.2
```

In order to access Sage globals, prefix them with `sage.:`:

```python
sage: f = cython_lambda('double x', 'sage.sin(x) + sage.a')
```

```python
sage: f(0)
Traceback (most recent call last):
... 
NameError: global 'a' is not defined
```

```python
sage: a = 25
sage: f(10)
24.45597888911063
```

```python
sage: a = 50
sage: f(10)
49.45597888911063
```

```python
>>> from sage.all import *

>>> f = cython_lambda('double x', 'sage.sin(x) + sage.a')

>>> f(Integer(0))
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

```python
Traceback (most recent call last):
...  
NameError: global 'a' is not defined
>>> a = Integer(25)
>>> f(Integer(10))
24.45597888911063
>>> a = Integer(50)
>>> f(Integer(10))
49.45597888911063
```

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'
```

```python
>>> from sage.all import *

>>> from sage.misc.cython import sanitize

>>> sanitize('abc')
'abc'

>>> sanitize('abc/def')
'abc_def'

>>> sanitize('123/def-hij/file.py')
'_123_def_hij_file_py'
```

3.2 Fortran compiler

```python
class sage.misc.inline_fortran.InlineFortran(globals=None)
    Bases: object

    add_library(s)

    add_library_path(s)

    eval(x, globals=None, locals=None)
        Compile fortran code x and adds the functions in it to globals.

        INPUT:
        • x – Fortran code
        • globals – a dict to which to add the functions from the fortran module
        • locals – ignored

        EXAMPLES:
```

360 Chapter 3. Code Evaluation
```python
# needs numpy

code = '''
....: C FILE: FIB1.F
....: SUBROUTINE FIB(A,N)
....: C
....: CALCULATE FIRST N FIBONACCI NUMBERS
....: C
....: INTEGER N
....: REAL*8 A(N)
....: DO I=1,N
....: IF (I.EQ.1) THEN
....: A(I) = 0.0D0
....: ELSEIF (I.EQ.2) THEN
....: A(I) = 1.0D0
....: ELSE
....: A(I) = A(I-1) + A(I-2)
....: ENDIF
....: ENDDO
....: END
....: C END FILE FIB1.F
....: '''

code = fortran(code, globals())

import numpy

a = numpy.array(range(10), dtype=float)

fib(a, 10)

a

array([ 0., 1., 1., 2., 3., 5., 8., 13., 21., 34.])
```
3.3 A parser for symbolic equations and expressions

It is both safer and more powerful than using Python’s eval, as one has complete control over what names are used (including dynamically creating variables) and how integer and floating point literals are created.

AUTHOR:

• Robert Bradshaw 2008-04 (initial version)

```python
class sage.misc.parser.LookupNameMaker:
    Bases: object

    This class wraps a dictionary as a callable for use in creating names. It takes a dictionary of names, and an (optional) callable to use when the given name is not found in the dictionary.

    EXAMPLES:

    sage: from sage.misc.parser import LookupNameMaker
    sage: maker = LookupNameMaker({pi: pi}, var)
    sage: maker(pi)
    pi
    sage: maker(pi) is pi
    True
    sage: maker('a')
    a
    >>> from sage.all import *
    >>> from sage.misc.parser import LookupNameMaker
    >>> maker = LookupNameMaker({pi: pi}, var)
    >>> maker(pi)
    pi
    >>> maker(pi) is pi
    True
    >>> maker('a')
    a
```

```python
set_names(new_names)
```

```python
class sage.misc.parser.Parser:
    Bases: object

    Create a symbolic expression parser.

    INPUT:

    • make_int – callable object to construct integers from strings (default int)
    • make_float – callable object to construct real numbers from strings (default float)
    • make_var – callable object to construct variables from strings (default str) this may also be a dictionary of variable names
    • make_function – callable object to construct callable functions from strings this may also be a dictionary
    • implicit_multiplication – whether or not to accept implicit multiplication

    OUTPUT:

    The evaluated expression tree given by the string, where the above functions are used to create the leaves of this tree.
EXAMPLES:

```python
sage: from sage.misc.parser import Parser
sage: p = Parser()
3
sage: p.parse("1+2 == 3")
True

sage: p = Parser(make_var=var)  # needs sage.symbolic
sage: p.parse("a*b^c - 3a")  # needs sage.symbolic
a*b^c - 3*a

sage: R.<x> = QQ[]

sage: p = Parser(make_var={x: x})

sage: p.parse("(x+1)^5-x")
x^5 + 5*x^4 + 10*x^3 + 10*x^2 + 4*x + 1

sage: p.parse("(x+1)^5-x").parent() is R
True

sage: p = Parser(make_float=RR, make_var=var,  # needs sage.symbolic
...: make_function={foo: lambda x: x*x+x})

sage: p.parse("1.5 + foo(b)")  # needs sage.symbolic
b^2 + b + 1.50000000000000

sage: p.parse("1.9").parent()  # needs sage.symbolic
Real Field with 53 bits of precision
```

```python
>>> from sage.all import *
>>> from sage.misc.parser import Parser
>>> p = Parser()
>>> p.parse("1+2")
3
>>> p.parse("1+2 == 3")
True

>>> p = Parser(make_var=var)  # needs sage.symbolic
...: make_function={foo: lambda x: x*x+x})

>>> p.parse("1.5 + foo(b)")  # needs sage.symbolic
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

\[ b^2 + b + 1.50000000000000 \]

```python
>>> p.parse("1.9").parent()
← sage.symbolic
Real Field with 53 bits of precision
```

**p_arg**(tokens)

Returns an expr, or a (name, expr) tuple corresponding to a single function call argument.

**EXAMPLES:**

Parsing a normal expression:

```python
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)  # needs sage.symbolic
sage: p.p_arg(Tokenizer("a+b"))  # needs sage.symbolic
a + b
```

A keyword expression argument:

```python
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)  # needs sage.symbolic
sage: p.p_arg(Tokenizer("val=a+b"))  # needs sage.symbolic
\(\text{val}, a + b\)
```

A lone list:

```python
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)  # needs sage.symbolic
sage: p.p_arg(Tokenizer("[x]"))  # needs sage.symbolic
[x]
```

**p_args**(tokens)

Returns a list, dict pair.

**EXAMPLES:**

```python
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser()

sage: p.p_args(Tokenizer("1,2,a=3"))
([[1, 2], {"a": 3}])

sage: p.p_args(Tokenizer("1, 2, a = 1+5^2"))
([[1, 2], {"a": 26}])
```

Chapter 3. Code Evaluation
>>> from sage.all import *
>>> from sage.misc.parser import Parser, Tokenizer
>>> p = Parser()
>>> p.p_args(Tokenizer("1,2,a=3"))
([1, 2], {'a': 3})
>>> p.p_args(Tokenizer("1, 2, a = 1+5^2"))
([1, 2], {'a': 26})

**p_atom(tokens)**

Parse an atom. This is either a parenthesized expression, a function call, or a literal name/int/float.

**EXAMPLES:**

```python
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var, make_function={sin: sin})
sage: p.p_atom(Tokenizer("1"))
1
sage: p.p_atom(Tokenizer("12"))
12
sage: p.p_atom(Tokenizer("12.5"))
12.5
sage: p.p_atom(Tokenizer("(1+a)"))
a + 1
sage: p.p_atom(Tokenizer("(1+a)^2"))
a + 1
sage: p.p_atom(Tokenizer("sin(1+a)"))
sin(a + 1)
sage: p = Parser(make_var=var,
....: make_function={'foo': sage.misc.parser.foo})
sage: p.p_atom(Tokenizer("foo(a, b, key=value)"))
((a, b), {'key': value})
sage: p.p_atom(Tokenizer("foo()"))
(()), ({})
```

---

### 3.3. A parser for symbolic equations and expressions

---
**p_eqn** *(tokens)*

Parse an equation or expression.

This is the top-level node called by the code `parse` function.

**EXAMPLES:**

```python
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)  # needs sage.symbolic
sage: p.p_eqn(Tokenizer("1+a"))  # needs sage.symbolic
a + 1
sage: p.p_eqn(Tokenizer("a == b"))  # needs sage.symbolic
a == b
sage: p.p_eqn(Tokenizer("a < b"))  # needs sage.symbolic
a < b
sage: p.p_eqn(Tokenizer("a > b"))  # needs sage.symbolic
a > b
sage: p.p_eqn(Tokenizer("a <= b"))  # needs sage.symbolic
a <= b
sage: p.p_eqn(Tokenizer("a >= b"))  # needs sage.symbolic
a >= b
sage: p.p_eqn(Tokenizer("a != b"))  # needs sage.symbolic
a != b
```

```python
>>> from sage.all import *
>>> from sage.misc.parser import Parser, Tokenizer
>>> p = Parser(make_var=var)  # needs sage.symbolic
>>> p.p_eqn(Tokenizer("1+a"))  # needs sage.symbolic
a + 1
```

**p_expr** *(tokens)*

Parse a list of one or more terms.

**EXAMPLES:**

```python
sage: # needs sage.symbolic
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)
```

(continues on next page)
Utilities, Release 10.4

sage: p.p_expr(Tokenizer("a+b"))
a + b

sage: p.p_expr(Tokenizer("a"))
a

sage: p.p_expr(Tokenizer("a - b + 4*c - d^2"))
-d^2 + a - b + 4*c

sage: p.p_expr(Tokenizer("a - 3"))
a + 3

sage: p.p_expr(Tokenizer("a + 1 == b"))
a + 1

>>> from sage.all import *
>>> # needs sage.symbolic
>>> from sage.misc.parser import Parser, Tokenizer

>>> p = Parser(make_var=var)
>>> p.p_expr(Tokenizer("a+b"))
a + b

>>> p.p_expr(Tokenizer("a"))
a

>>> p.p_expr(Tokenizer("a - b + 4*c - d^2"))
-d^2 + a - b + 4*c

>>> p.p_expr(Tokenizer("a - 3"))
a + 3

>>> p.p_expr(Tokenizer("a + 1 == b"))
a + 1

p_factor (tokens)

Parse a single factor, which consists of any number of unary +/- and a power.

EXAMPLES:

sage: from sage.misc.parser import Parser, Tokenizer

sage: R.<t> = ZZ[[t]]

sage: p = Parser(make_var={t: t})

sage: p.p_factor(Tokenizer("- -t"))
t

sage: p.p_factor(Tokenizer("- + - -t^2"))
-t^2

sage: p.p_factor(Tokenizer("t^11 * x"))
t^11

>>> from sage.all import *

>>> from sage.misc.parser import Parser, Tokenizer

>>> R = ZZ[['t'']]; (t,) = R._first_ngens(1)

>>> p = Parser(make_var={'t': t})

>>> p.p_factor(Tokenizer("- -t"))
t

>>> p.p_factor(Tokenizer("- + -t^2"))
-t^2

>>> p.p_factor(Tokenizer("t^11 * x"))
t^11

p_list (tokens)

Parse a list of items.

EXAMPLES:

3.3. A parser for symbolic equations and expressions
```python
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)  # needs sage.symbolic
sage: p.p_list(Tokenizer("[1+2, 1e3]"))  # needs sage.symbolic
[3, 1000.0]
sage: p.p_list(Tokenizer("[]"))  # needs sage.symbolic
[]
```

```python
>>> from sage.all import *
>>> from sage.misc.parser import Parser, Tokenizer
>>> p = Parser(make_var=var)  # needs sage.symbolic
>>> p.p_matrix(Tokenizer("([a,0],[0,a])"))  # needs sage.symbolic
[a 0]
[0 a]
```

### p_matrix(tokens)

Parse a matrix

**EXAMPLES:**

```python
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)  # needs sage.symbolic
sage: p.p_matrix(Tokenizer("([a,0],[0,a])"))  # needs sage.symbolic
[a 0]
[0 a]
```

```python
>>> from sage.all import *
>>> from sage.misc.parser import Parser, Tokenizer
>>> p = Parser(make_var=var)  # needs sage.symbolic
>>> p.p_matrix(Tokenizer("([a,0],[0,a])"))  # needs sage.symbolic
[a 0]
[0 a]
```

### p_power(tokens)

Parses a power. Note that exponentiation groups right to left.

**EXAMPLES:**

```python
sage: from sage.misc.parser import Parser, Tokenizer
sage: R.<t> = ZZ[[t]]
sage: p = Parser(make_var=('t': t))
sage: p.p_factor(Tokenizer("-(1+t)^-1"))
-1 + t - t^2 + t^3 - t^4 + t^5 - t^6 + t^7 - t^8 + t^9 - t^10 + t^11 - t^12 +
- t^13 - t^14 - t^15 + t^16 + t^17 + t^18 + t^19 + O(t^20)
sage: p.p_factor(Tokenizer("2^3^2")) == 2^9
(continues on next page)
```
Utilities, Release 10.4

(continued from previous page)

True
sage: # needs sage.symbolic
sage: p = Parser(make_var=var)
sage: p.p_factor(Tokenizer( x! ))
factorial(x)
sage: p.p_factor(Tokenizer( (x^2)! ))
factorial(x^2)
sage: p.p_factor(Tokenizer( x!^2 ))
factorial(x)^2
>>> from sage.all import *
>>> from sage.misc.parser import Parser, Tokenizer
>>> R = ZZ[[ t ]]; (t,) = R._first_ngens(1)
>>> p = Parser(make_var={ t : t})
>>> p.p_factor(Tokenizer("-(1+t)^-1"))
-1 + t - t^2 + t^3 - t^4 + t^5 - t^6 + t^7 - t^8 + t^9 - t^10 + t^11 - t^12 +␣
˓→t^13 - t^14 + t^15 - t^16 + t^17 - t^18 + t^19 + O(t^20)
>>> p.p_factor(Tokenizer("t**2"))
t^2
>>> p.p_power(Tokenizer("2^3^2")) == Integer(2)**Integer(9)
True
>>> # needs sage.symbolic
>>> p = Parser(make_var=var)
>>> p.p_factor(Tokenizer( x! ))
factorial(x)
>>> p.p_factor(Tokenizer( (x^2)! ))
factorial(x^2)
>>> p.p_factor(Tokenizer( x!^2 ))
factorial(x)^2

p_sequence(tokens)
Parse a (possibly nested) set of lists and tuples.
EXAMPLES:
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)
˓→needs sage.symbolic
sage: p.p_sequence(Tokenizer("[1+2,0]"))
˓→needs sage.symbolic
[[3, 0]]
sage: p.p_sequence(Tokenizer("(1,2,3) , [1+a, 2+b, (3+c), (4+d,)]"))
˓→needs sage.symbolic
[(1, 2, 3), [a + 1, b + 2, c + 3, (d + 4,)]]
>>> from sage.all import *
>>> from sage.misc.parser import Parser, Tokenizer
>>> p = Parser(make_var=var)
˓→needs sage.symbolic
>>> p.p_sequence(Tokenizer("[1+2,0]"))
˓→needs sage.symbolic
[[3, 0]]
>>> p.p_sequence(Tokenizer("(1,2,3) , [1+a, 2+b, (3+c), (4+d,)]"))
˓→needs sage.symbolic

#␣
#␣

#␣

#␣
#␣

#␣

(continues on next page)

3.3. A parser for symbolic equations and expressions

369


p_term (tokens)

Parse a single term (consisting of one or more factors).

EXAMPLES:

```python
sage: # needs sage.symbolic
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)
sage: p.p_term(Tokenizer("a*b"))
a*b
sage: p.p_term(Tokenizer("a * b / c * d"))
a*b*d/c
sage: p.p_term(Tokenizer("-a * b + c"))
-a*b
sage: p.p_term(Tokenizer("a*(b-c)^2"))
a*(b - c)^2
sage: p.p_term(Tokenizer("-3a"))
-3*a
```

```python
>>> from sage.all import *
>>> from sage.misc.parser import Parser, Tokenizer

p_tuple (tokens)

Parse a tuple of items.

EXAMPLES:

```python
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)

sage: p.p_tuple(Tokenizer("( (), (1), (1,), (1,2), (1,2,3), (1+2)^2, )"))
((), 1, (1,), (1, 2), (1, 2, 3), 9)
```

```python
>>> from sage.all import *
>>> from sage.misc.parser import Parser, Tokenizer

>>> p = Parser(make_var=var)

>>> p.p_tuple(Tokenizer("( (), (1), (1,), (1,2), (1,2,3), (1+2)^2, )"))
((), 1, (1,), (1, 2), (1, 2, 3), 9)
```
**parse** *(s, accept_eqn=True)*

Parse the given string.

EXAMPLES:

```python
sage: from sage.misc.parser import Parser
sage: p = Parser(make_var=var)  # needs sage.symbolic
sage: p.parse("E = m c^2")  # needs sage.symbolic
E == c^2*m
```

**parse_expression** *(s)*

Parse an expression.

EXAMPLES:

```python
sage: from sage.misc.parser import Parser
sage: p = Parser(make_var=var)  # needs sage.symbolic
sage: p.parse_expression('a-3b^2')  # needs sage.symbolic
-3*b^2 + a
```

**parse_sequence** *(s)*

Parse a (possibly nested) set of lists and tuples.

EXAMPLES:

```python
sage: # needs sage.symbolic
sage: from sage.misc.parser import Parser
sage: p = Parser(make_var=var)  # needs sage.symbolic
sage: p.parse_sequence("1,2,3")
[1, 2, 3]
sage: p.parse_sequence("[1,2,\(a,b,c+d\)]")
[1, 2, (a, b, c + d)]
sage: p.parse_sequence("13")
13
```

(continues on next page)
class sage.misc.parser.Tokenizer

Bases: object

This class takes a string and turns it into a list of tokens for use by the parser.

The tokenizer wraps a string object, to tokenize a different string create a new tokenizer.

EXAMPLES:

```python
>>> from sage.all import *
>>> from sage.misc.parser import Tokenizer

sage: Tokenizer("1.5+2^3-4-sin(x)").test()
[\texttt{FLOAT}(1.5), +, INT(2), ^, INT(3), -, NAME(sin), '(', \rightarrow NAME(x), ')']
```

The single character tokens are given by:

```python
>>> from sage.all import *
>>> from sage.misc.parser import Tokenizer

sage: Tokenizer("+-*/^(),=<\[
\]
{\}").test()
[+, -, *, /, ^, (, ), ,, =, <, >, [\texttt{\textbackslash [} \texttt{\textbackslash ]}, \texttt{\textbackslash \{} \texttt{\textbackslash \}}]
```

Two-character comparisons accepted are:

```python
>>> from sage.all import *
>>> from sage.misc.parser import Tokenizer

sage: Tokenizer("<= >= != == **").test()
[\texttt{LESSEQ}, \texttt{GREATEREQ}, \texttt{NOTEQ}, '=', '^']
```

Integers are strings of 0-9:

```python
>>> from sage.all import *
>>> from sage.misc.parser import Tokenizer

sage: Tokenizer("1 123 9879834759873452908375013").test()
[\texttt{INT}(1), \texttt{INT}(123), \texttt{INT}(9879834759873452908375013)]
```

```python
>>> from sage.misc.parser import Parser
>>> p = Parser(make_var=var)

>>> p.parse_sequence("1,2,3")
[1, 2, 3]

>>> p.parse_sequence("[1,2,(a,b,c+d)]")
[1, 2, (a, b, c + d)]

>>> p.parse_sequence("13")
13
```
Floating point numbers can contain a single decimal point and possibly exponential notation:

```python
sage: Tokenizer("1. .01 1e3 1.e-3").test()
['FLOAT(1.)', 'FLOAT(.01)', 'FLOAT(1e3)', 'FLOAT(1.e-3)']
```

```python
>>> from sage.all import *
```

```python
>>> Tokenizer("1. .01 1e3 1.e-3").test()
['FLOAT(1.)', 'FLOAT(.01)', 'FLOAT(1e3)', 'FLOAT(1.e-3)']
```

Note that negative signs are not attached to the token:

```python
sage: Tokenizer("-1 -1.2").test()
['-', 'INT(1)', '-', 'FLOAT(1.2)']
```

```python
>>> from sage.all import *
```

```python
>>> Tokenizer("-1 -1.2").test()
['-', 'INT(1)', '-', 'FLOAT(1.2)']
```

Names are alphanumeric sequences not starting with a digit:

```python
sage: Tokenizer("a a1 _a_24").test()
['NAME(a)', 'NAME(a1)', 'NAME(_a_24)']
```

```python
>>> from sage.all import *
```

```python
>>> Tokenizer("a a1 _a_24").test()
['NAME(a)', 'NAME(a1)', 'NAME(_a_24)']
```

There is special handling for matrices:

```python
sage: Tokenizer("matrix(a)").test()
['MATRIX', '(', 'NAME(a)', ')']
```

```python
>>> from sage.all import *
```

```python
>>> Tokenizer("matrix(a)").test()
['MATRIX', '(', 'NAME(a)', ')']
```

Anything else is an error:

```python
sage: Tokenizer("@~").test()
['ERROR', 'ERROR', 'ERROR']
```

```python
>>> from sage.all import *
```

```python
>>> Tokenizer("@~").test()
['ERROR', 'ERROR', 'ERROR']
```

No attempt for correctness is made at this stage:

```python
sage: Tokenizer("( 5e5) ").test()
['(', ')', '(', 'FLOAT(5e5)', 'NAME(e5)']
```

```python
sage: Tokenizer("@%").test()
['ERROR', 'ERROR', 'ERROR']
```

```python
>>> from sage.all import *
```

```python
>>> Tokenizer("( 5e5) ").test()
['(', ')', '(', 'FLOAT(5e5)', 'NAME(e5)']
```

(continues on next page)
backtrack()

Put self in such a state that the subsequent call to next() will return the same as if next() had not been called.

Currently, one can only backtrack once.

EXAMPLES:

```python
sage: from sage.misc.parser import Tokenizer, token_to_str
sage: t = Tokenizer("a+b")
sage: token_to_str(t.next())
'NAME'
sage: t.backtrack()  # the return type is bint for performance reasons
False
sage: token_to_str(t.next())
'+'
```

```python
>>> from sage.all import *
>>> from sage.misc.parser import Tokenizer, token_to_str

```
Utilities, Release 10.4

(continued from previous page)

```python
>>> token_to_str(t.next())
'NAME'
```

```python
>>> token_to_str(t.last())
'NAME'
```

last_token_string()

Return the actual contents of the last token.

EXAMPLES:

```python
sage: from sage.misc.parser import Tokenizer, token_to_str
sage: t = Tokenizer("a - 1e5")
```

```python
sage: token_to_str(t.next())
'NAME'
```

```python
sage: t.last_token_string()
'a'
```

```python
sage: token_to_str(t.next())
'-'
```

```python
sage: token_to_str(t.next())
'FLOAT'
```

```python
sage: t.last_token_string()
'1e5'
```

```python
>>> from sage.all import *
```

```python
>>> from sage.misc.parser import Tokenizer, token_to_str
```

```python
>>> t = Tokenizer("a - 1e5")
```

```python
>>> token_to_str(t.next())
'NAME'
```

```python
>>> t.last_token_string()
'a'
```

```python
>>> token_to_str(t.next())
'-'
```

```python
>>> token_to_str(t.next())
'FLOAT'
```

```python
>>> t.last_token_string()
'1e5'
```

next()

Returns the next token in the string.

EXAMPLES:

```python
sage: from sage.misc.parser import Tokenizer, token_to_str
sage: t = Tokenizer("a+3")
```

```python
sage: token_to_str(t.next())
'NAME'
```

```python
sage: token_to_str(t.next())
'+'
```

```python
sage: token_to_str(t.next())
'INT'
```

```python
sage: token_to_str(t.next())
'EOS'
```

```python
>>> from sage.all import *
```

```python
>>> from sage.misc.parser import Tokenizer, token_to_str
```

```python
>>> t = Tokenizer("a+3")
```

(continues on next page)
peek()

Returns the next token that will be encountered, without changing the state of self.

EXAMPLES:

```python
sage: from sage.misc.parser import Tokenizer, token_to_str
sage: t = Tokenizer("a+b")
```

```python
sage: token_to_str(t.peek())
'NAME'
```

```python
sage: token_to_str(t.next())
'+'
```

```python
sage: token_to_str(t.peek())
'NAME'
```

```python
sage: token_to_str(t.next())
'NAME'
```

```python
sage: token_to_str(t.peek())
'+'
```

```python
sage: token_to_str(t.peek())
'+'
```

```python
sage: token_to_str(t.next())
'+'
```

reset(pos=0)

Reset the tokenizer to a given position.

EXAMPLES:

```python
sage: from sage.misc.parser import Tokenizer
sage: t = Tokenizer("a+b*c")
```

```python
sage: t.test()
['NAME(a)', '+', 'NAME(b)', '*', 'NAME(c)']
```

```python
sage: t.test()
[]
```

```python
sage: t.reset()
```

```python
sage: t.test()
[]
```

```python
sage: t.reset(3)
```

```python

376 Chapter 3. Code Evaluation
```
sage: t.test()
['*', 'NAME(c)']

```python
>>> from sage.all import *
>>> from sage.misc.parser import Tokenizer
>>> t = Tokenizer("a+b*c")
>>> t.test()
['NAME(a)', '+', 'NAME(b)', '*', 'NAME(c)']
>>> t.reset()
[]
>>> t.test()
['NAME(a)', '+', 'NAME(b)', '*', 'NAME(c)']
>>> t.reset(Integer(3))
>>> t.test()
['*', 'NAME(c)']
```

No care is taken to make sure we don’t jump in the middle of a token:

```python
sage: t = Tokenizer("12345+a")
sage: t.test()
['INT(12345), +, NAME(a)']
sage: t.reset(2)
sage: t.test()
['INT(345), +, NAME(a)']
```

```python
>>> from sage.all import *
>>> from sage.misc.parser import Tokenizer
>>> t = Tokenizer("a b 3")
>>> t.test()
['NAME(a), NAME(b), INT(3)']
>>> t.reset()
[]
```

`test()`

This is a utility function for easy testing of the tokenizer.

Destructively read off the tokens in self, returning a list of string representations of the tokens.

**EXAMPLES:**

```python
sage: from sage.misc.parser import Tokenizer
sage: t = Tokenizer("a b 3")
sage: t.test()
['NAME(a), NAME(b), 'INT(3)']
```

```python
>>> from sage.all import *
>>> from sage.misc.parser import Tokenizer
>>> t = Tokenizer("a b 3")
>>> t.test()
['NAME(a), NAME(b), 'INT(3)']
>>> t.test()
[]
```
This is a function for testing that simply returns the arguments and keywords passed into it.

**EXAMPLES:**

```python
sage: from sage.misc.parser import foo
sage: foo(1, 2, a=3)
((1, 2), {'a': 3})
```

For speed reasons, tokens are integers. This function returns a string representation of a given token.

**EXAMPLES:**

```python
sage: from sage.misc.parser import Tokenizer, token_to_str
sage: t = Tokenizer("+ 2")
sage: token_to_str(t.next())
'+'
sage: token_to_str(t.next())
'INT'
```

### 3.4 Evaluating Python code without any preparsing

**class** `sage.misc.python.Python`

Bases: `object`

 Allows for evaluating a chunk of code without any preparsing.

**eval** *(x, globals, locals=None)*

Evaluate `x` with given `globals`; `locals` is completely ignored.

This is specifically meant for evaluating code blocks with `%python` in the notebook.

**INPUT:**

- `x` – a string
- `globals` – a dictionary
- `locals` – completely IGNORED

**EXAMPLES:**
sage: from sage.misc.python import Python
sage: python = Python()
Output: 4

```
>>> from sage.all import *
>>> from sage.misc.python import Python
>>> python = Python()
>>> python.eval(2+2, globals())
Output: 4
```

Any variables that are set during evaluation of the block will propagate to the globals dictionary.

```
sage: python.eval('a=5\nb=7\na+b', globals())
Output: 12

>>> from sage.all import *
>>> python.eval('a=5\nb=7\na+b', globals())
Output: 12
>>> b
Output: 7
```

The locals variable is ignored – it is there only for completeness. It is ignored since otherwise the following will not work:

```
sage: python.eval("def foo():
  return 'foo'\nprint(foo())\ndef mumble():
  print('mumble {}\n  print(mumble.format(foo()))")", globals())
foo
mumble foo

```
```
sage: python.eval("def foo():
  return 'foo'\nprint(foo())\ndef mumble():
  print('mumble {}\n  print(mumble.format(foo()))")", globals())
foo
mumble foo

```
```
>>> from sage.all import *
>>> python.eval("def foo():
  return 'foo'\nprint(foo())\ndef mumble():
  print('mumble {}\n  print(mumble.format(foo()))")", globals())
foo
mumble foo
```
```
>>> mumble
<function mumble at ...>
```
```
```
```
```
3.5 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, parse 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)')
```

(continues on next page)
This example illustrates setting a variable for use in evaluation.

```python
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:

```python
sage: sage_eval('p', cmds='R.<x> = QQ[]
p = x^2 + 1')
x^2 + 1
```

```python
>>> from sage.all import *
>>> bernoulli = lambda x : x**Integer(2)
>>> bernoulli(Integer(6))
36
>>> eval('bernoulli(6)')
36
>>> sage_eval('bernoulli(6)')
# needs sage.libs.flint
```

```python
sage: QQ = lambda x : x^2
sage: QQ(2)
4
sage: sage_eval(QQ(2))
2
sage: parent(sage_eval('QQ(2)'))
Rational Field
```

```python
>>> from sage.all import *
>>> QQ = lambda x : x**Integer(2)
>>> QQ(Integer(2))
4
>>> sage_eval(QQ(2))
2
>>> parent(sage_eval('QQ(2)'))
Rational Field
```

You can also specify a sequence of commands to be run before the expression is evaluated:

```python
sage: sage_eval('p', cmds='R.<x> = QQ[]
p = x^2 + 1')
x^2 + 1
```

```python
>>> from sage.all import *
>>> sage_eval('p', cmds='R.<x> = QQ[]
p = 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:

```python
sage: vars = {}
```

```python
sage: sage_eval('None', cmds='y = 3', locals=vars)
```

```python
sage: vars['y'], parent(vars['y'])
```

(3, Integer Ring)

```python
>>> from sage.all import *
```

```python
>>>
```

```python
>>>
```

```python
>>>
```

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.)

```python
sage: sage_eval(('f(x) = x^2', 'f(3)'))
```

# needs sage.symbolic

```python
sage: vars = {'rt2': sqrt(2.0)}
```

```python
sage: sage_eval(('rt2 += 1', 'rt2', vars))
```

2.41421356237309

```python
sage: vars['rt2']
```

1.41421356237310

```python
>>> from sage.all import *
```

```python
>>>
```

```python
>>>
```

```python
>>>
```

This example illustrates how `sage_eval` can be useful when evaluating the output of other computer algebra systems.

```python
sage: # needs sage.libs.gap
```

```python
sage: R.<x> = PolynomialRing(RationalField())
```

```python
sage: gap.eval(R:=PolynomialRing(Rationals,"x");)
```

```python
Rationals[x]
```

```python
sage: ff = gap.eval('x:=IndeterminatesOfPolynomialRing(R);; f:=x^2+1;'); ff
```

```python
'x^2+1'
```

```python
sage: sage_eval(ff, locals={'x':x})
```

```python
x^2 + 1
```

```python
sage: eval(ff)
```

Traceback (most recent call last):
...

RuntimeError: Use ** for exponentiation, not '^', which means xor in Python, and has the wrong precedence.

```python
>>> from sage.all import *
```

```python
>>>
```

(continues on next page)
Here you can see that `eval()` simply will not work but `sage_eval()` will.

```
sage.misc.sage_eval.sageobj(x, vars=None)
```

Return a native Sage object associated to `x`, if possible and implemented.

If the object has a `_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'))))  # needs sage.libs.pari
<class 'sage.rings.rational.Rational'>
```

```
sage: n = 5/2
sage: sageobj(n) is n
True
```

```
sage: k = sageobj('2^3/1', {'2':ZZ}); k
512
sage: type(k)
<class 'sage.rings.integer.Integer'>
```

```
>> from sage.all import *
>> type(sageobj(gp('34/56'))))  # needs sage.libs.pari
<class 'sage.rings.rational.Rational'>
```

```
>> n = Integer(5)/Integer(2)
>> sageobj(n) is n
True
>> k = sageobj('2^3/1', {'2':ZZ}); k
512
>> type(k)
<class 'sage.rings.integer.Integer'>
```

This illustrates interfaces:

```
sage: # needs sage.libs.pari
sage: f = gp('2/3')
```

```
sage: f._sage_()
```

(continues on next page)
2/3
sage: type(f._sage_())
<class 'sage.rings.rational.Rational'>

sage: # needs sage.libs.gap
sage: a = gap(939393/2433)
sage: a._sage_()
313131/811
sage: type(a._sage_())
<class 'sage.rings.rational.Rational'>

>>> from sage.all import *
>>> # needs sage.libs.pari
>>> f = gp(2/3)
>>> type(f)
<class 'sage.interfaces.gp.GpElement'>
>>> f._sage_()
2/3
>>> type(f._sage_())
<class 'sage.rings.rational.Rational'>

3.6 Evaluating shell scripts

class sage.misc.sh.Sh
    Bases: object

    Evaluates a shell script and returns the output.
    To use this from the notebook type sh at the beginning of the input cell. The working directory is then the (usually temporary) directory where the Sage worksheet process is executing.

    eval (code, globals=None, locals=None)

    This is difficult to test because the output goes to the screen rather than being captured by the doctest program, so the following really only tests that the command doesn’t bomb, not that it gives the right output:

sage: sh.eval('''echo "Hello there" if [ $? -eq 0 ]; then\necho "good"\nfi''')  # random output

>>> from sage.all import *
>>> sh.eval('''echo "Hello there" if [ $? -eq 0 ]; then\necho "good"\nfi''')  # random output
4.1 Formatted Output

4.1.1 Symbols for Character Art

This module defines single- and multi-line symbols.

EXAMPLES:

```python
sage: from sage.typeset.symbols import *
sage: symbols = ascii_art('')
sage: for i in range(1, 5):
    ....:    symbols += ascii_left_parenthesis.character_art(i)
    ....:    symbols += ascii_art(' ')  # Added a space for clarity
    ....:    symbols += ascii_right_parenthesis.character_art(i)
    ....:    symbols += ascii_art(' ')  # Added a space for clarity
sage: for i in range(1, 5):
    ....:    symbols += ascii_left_square_bracket.character_art(i)
    ....:    symbols += ascii_art(' ')  # Added a space for clarity
    ....:    symbols += ascii_right_square_bracket.character_art(i)
    ....:    symbols += ascii_art(' ')  # Added a space for clarity
sage: for i in range(1, 5):
    ....:    symbols += ascii_left_curly_brace.character_art(i)
    ....:    symbols += ascii_art(' ')  # Added a space for clarity
    ....:    symbols += ascii_right_curly_brace.character_art(i)
    ....:    symbols += ascii_art(' ')  # Added a space for clarity
sage: symbols
( ) [ ] { }
```

```
( ) ( ) ( ) [ ] [ ] [ ] { } { } { }
```

```
( ) ( ) ( ) ( ) [ ] [ ] [ ] [ ] { } { } { } { }
```

sage: symbols = unicode_art('')
sage: for i in range(1, 5):
    ....:    symbols += unicode_left_parenthesis.character_art(i)
    ....:    symbols += unicode_art(' ')  # Added a space for clarity
    ....:    symbols += unicode_right_parenthesis.character_art(i)
    ....:    symbols += unicode_art(' ')  # Added a space for clarity
sage: for i in range(1, 5):
    ....:    symbols += unicode_left_square_bracket.character_art(i)
    ....:    symbols += unicode_art(' ')  # Added a space for clarity
    ....:    symbols += unicode_right_square_bracket.character_art(i)
    ....:    symbols += unicode_art(' ')  # Added a space for clarity
sage: for i in range(1, 5):
    ....:    symbols += unicode_left_curly_brace.character_art(i)
    ....:    symbols += unicode_art(' ')  # Added a space for clarity
    ....:    symbols += unicode_right_curly_brace.character_art(i)
    ....:    symbols += unicode_art(' ')  # Added a space for clarity
```
Utilities, Release 10.4

sage: for i in range(1, 5):
    symbols += unicode_left_curly_brace.character_art(i)
    symbols += unicode_art(' ')
    symbols += unicode_right_curly_brace.character_art(i)
    symbols += unicode_art(' ')

sage: symbols

( ) [ ] { }

sage: symbols = ascii_art()  

>>> from sage.all import *
>>> from sage.typeset.symbols import *

>>> symbols = ascii_art('')

>>> for i in range(Integer(1), Integer(5)):
    symbols += ascii_left_parenthesis.character_art(i)
    symbols += ascii_art(' ')
    symbols += ascii_right_parenthesis.character_art(i)
    symbols += ascii_art(' ')

>>> symbols = unicode_art()  

>>> for i in range(Integer(1), Integer(5)):
    symbols += unicode_left_curly_brace.character_art(i)
    symbols += unicode_art(' ')
    symbols += unicode_right_curly_brace.character_art(i)
    symbols += unicode_art(' ')

>>> symbols

( ) ( ) ( ) [ ] [ ] [ ] { } { } { }

>>>

Chapter 4. Formatted Output
class sage.typeset_symbols.CompoundAsciiSymbol(character, top, extension, bottom, middle=None, middle_top=None, middle_bottom=None, top_2=None, bottom_2=None):

Bases: CompoundSymbol

class_art(num_lines)

Return the ASCII art of the symbol

EXAMPLES:

sage: from sage.typeset_symbols import *
sage: ascii_left_curly_brace.character_art(3)
{
{

>>> from sage.all import *
>>> from sage.typeset_symbols import *
>>> ascii_left_curly_brace.character_art(Integer(3))
{
{

class sage.typeset_symbols.CompoundSymbol(character, top, extension, bottom, middle=None, middle_top=None, middle_bottom=None, top_2=None, bottom_2=None):

Bases: SageObject

A multi-character (ascii/unicode art) symbol

INPUT:

Instead of string, each of these can be unicode in Python 2:

- character – string. The single-line version of the symbol.
- top – string. The top line of a multi-line symbol.
- extension – string. The extension line of a multi-line symbol (will be repeated).
- middle – optional string. The middle part, for example in curly braces. Will be used only once for the symbol, and only if its height is odd.
- middle_top – optional string. The upper half of the 2-line middle part if the height of the symbol is even. Will be used only once for the symbol.
- middle_bottom – optional string. The lower half of the 2-line middle part if the height of the symbol is even. Will be used only once for the symbol.
- top_2 – optional string. The upper half of a 2-line symbol.
- bottom_2 – optional string. The lower half of a 2-line symbol.

EXAMPLES:

sage: from sage.typeset_symbols import CompoundSymbol
sage: i = CompoundSymbol('I', '+', '|', '+', '|')
sage: i.print_to_stdout(1)
I
sage: i.print_to_stdout(3)
+ |
+

```python
>>> from sage.all import *
>>> from sage.typeset.symbols import CompoundSymbol
>>> i = CompoundSymbol('I', '+', '|', '+', '|')
>>> i.print_to_stdout(Integer(1))
I
>>> i.print_to_stdout(Integer(3))
```

**print_to_stdout** (*num_lines*)

Print the multi-line symbol

This method is for testing purposes.

**INPUT:**

- *num_lines* – integer. The total number of lines.

**EXAMPLES:**

```python
sage: from sage.typeset.symbols import *
sage: unicode_integral.print_to_stdout(1)
∫
```

```python
sage: unicode_integral.print_to_stdout(2)
⌠
```

```python
sage: unicode_integral.print_to_stdout(3)
⎮
```

```python
sage: unicode_integral.print_to_stdout(4)
⎮
```

```python
>>> from sage.all import *
>>> from sage.typeset.symbols import *
>>> unicode_integral.print_to_stdout(Integer(1))
∫
```

```python
>>> unicode_integral.print_to_stdout(Integer(2))
⌠
```

```python
>>> unicode_integral.print_to_stdout(Integer(3))
⎮
```

```python
>>> unicode_integral.print_to_stdout(Integer(4))
⎮
```
4.1.2 Base Class for Character-Based Art

This is the common base class for sage.typeset.ascii_art.AsciiArt and sage.typeset.unicode_art.UnicodeArt. They implement simple graphics by placing characters on a rectangular grid, in other words, using monospace fonts. The difference is that one is restricted to 7-bit ascii, the other uses all unicode code points.

class sage.typeset.character_art.CharacterArt(lines=[], breakpoints=[], baseline=None)

Bases: SageObject

Abstract base class for character art

INPUT:

- lines – the list of lines of the representation of the character art object
- breakpoints – the list of points where the representation can be split
- baseline – the reference line (from the bottom)

Instead of just integers, breakpoints may also contain tuples consisting of an offset and the breakpoints of a nested substring at that offset. This is used to prioritize the breakpoints, as line breaks inside the substring will be avoided if possible.

EXAMPLES:

```
sage: i = var('i')  # ...

"""needs sage.symbolic"

sage: ascii_art(sum(pi^i/ factorial(i) * x^i, i, 0, oo))  # ...
```

(continues on next page)
```python
needs sage.symbolic
pi*x
e
```

```python
>>> from sage.all import *
>>> i = var('i')  # needs sage.symbolic
>>> ascii_art(sum(pi**i/factorial(i)*x**i, i, Integer(0), oo))  # needs sage.symbolic
pi*x
e
```

classmethod empty()

Return the empty character art object

EXAMPLES:

```python
sage: from sage.typeset.ascii_art import AsciiArt
sage: AsciiArt.empty()
```

```python
>>> from sage.all import *
>>> from sage.typeset.ascii_art import AsciiArt
>>> AsciiArt.empty()
```

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.

get_breakpoints()

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

This method is deprecated, as its output is an implementation detail. The mere breakpoints of a character art element do not reflect the best way to split it if nested structures are involved. For details, see Issue #29204.

For example the expression:

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

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

EXAMPLES:

```python
sage: from sage.typeset.ascii_art import AsciiArt
sage: p3 = AsciiArt([' * ', '***'])
sage: p5 = AsciiArt([' * ', ' * * ', '*****'])
```
Utilities, Release 10.4

(continued from previous page)

```python
sage: aa = ascii_art([p3, p5])
sage: aa.get_breakpoints()
doctest:...: DeprecationWarning: get_breakpoints() is deprecated
See https://github.com/sagemath/sage/issues/29204 for details.
[6]
```

```python
>>> from sage.all import *
>>> from sage.typeset.ascii_art import AsciiArt
>>> p3 = AsciiArt([' * ', '***'])
>>> p5 = AsciiArt([' * ', ' * * ', '*****'])
>>> aa = ascii_art([p3, p5])
>>> aa.get_breakpoints()
doctest:...: DeprecationWarning: get_breakpoints() is deprecated
See https://github.com/sagemath/sage/issues/29204 for details.
[6]
```
width()

Return the length (width) of the ASCII art object.

OUTPUT:

Integer. The number of characters in each line.

### 4.1.3 Factory for Character-Based Art

```python
class sage.typeset.character_art_factory.CharacterArtFactory(
    art_type, string_type, magic_method_name, parenthesis, square_bracket, curly_brace
):
```

**Bases:** `SageObject`

Abstract base class for character art factory

This class is the common implementation behind `ascii_art()` and `unicode_art()`.

**INPUT:**

- `art_type` – type of the character art (i.e. a subclass of `CharacterArt`)
- `string_type` – type of strings (the lines in the character art, e.g. `str` or `unicode`).
- `magic_method_name` – name of the Sage magic method (e.g. `'_ascii_art_'` or `'_unicode_art_'`).
- `parenthesis` – left/right pair of two multi-line symbols. The parenthesis, a.k.a. round brackets (used for printing tuples).
- `square_bracket` – left/right pair of two multi-line symbols. The square brackets (used for printing lists).
- `curly_brace` – left/right pair of two multi-line symbols. The curly braces (used for printing sets).

**EXAMPLES:**

```python
sage: from sage.typeset.ascii_art import _ascii_art_factory as factory
sage: type(factory)
<class 'sage.typeset.character_art_factory.CharacterArtFactory'>

>>> from sage.all import *
>>> from sage.typeset.ascii_art import _ascii_art_factory as factory
>>> type(factory)
<class 'sage.typeset.character_art_factory.CharacterArtFactory'>
```

**build**(obj, baseline=None)

Construct a character art representation.

**INPUT:**

- `obj` – anything; the object whose ascii art representation we want
- `baseline` – (optional) the baseline of the object

**OUTPUT:**

Character art object.

**EXAMPLES:**
sage: result = ascii_art(integral(exp(x+x^2)/(x+1), x))  # needs sage.symbolic
...
sage: result  # needs sage.symbolic
...

>>> from sage.all import *

```python
from sage.all import *
result = ascii_art(integral(exp(x+x**Integer(2))/(x+Integer(1)), x))  # needs sage.symbolic
```

**build_container** (*content*, *left_border*, *right_border*, *baseline=0*)

Return character art for a container.

**INPUT:**

- *content* - `CharacterArt`; the content of the container, usually comma-separated entries
- *left_border* - `CompoundSymbol`; the left border of the container
- *right_border* - `CompoundSymbol`; the right border of the container
- *baseline* - (default: 0) the baseline of the object

**build_dict** (*d*, *baseline=0*)

Return a character art output of a dictionary.

**build_empty** ()

Return the empty character art object

**OUTPUT:**

Character art instance.

**EXAMPLES:**

```python
sage: from sage.typeset.ascii_art import _ascii_art_factory as factory
sage: str(factory.build_empty())
```

---

4.1. Formatted Output
```python
>>> from sage.all import *
>>> from sage.typeset.ascii_art import _ascii_art_factory as factory
>>> str(factory.build_empty())
```

**build_from_magic_method** *(obj, baseline=None)*

Return the character art object created by the object’s magic method

**OUTPUT:**

Character art instance.

**EXAMPLES:**

```
sage: from sage.typeset.ascii_art import _ascii_art_factory as factory
sage: out = factory.build_from_magic_method(identity_matrix(2)); out  # needs sage.modules
[[1 0]
 [0 1]]
sage: type(out)  # needs sage.modules
<class 'sage.typeset.ascii_art.AsciiArt'>
```

```python
>>> out = factory.build_from_magic_method(identity_matrix(Integer(2))); out
a
bb
ccc
```

```python
>>> type(out)  # needs sage.modules
<class 'sage.typeset.ascii_art.AsciiArt'>
```

**build_from_string** *(obj, baseline=0)*

Return the character art object created from splitting the object’s string representation.

**INPUT:**

- obj – utf-8 encoded byte string or unicode
- baseline – (default: 0) the baseline of the object

**OUTPUT:**

Character art instance.

**EXAMPLES:**

```
sage: out = factory.build_from_string('a
bb
ccc')
sage: out + out + out
a
bb
ccc
```

```python
>>> out = factory.build_from_string('a\nbb\ncccc'); out + out + out
```

```python
>>> type(out)  # needs sage.modules
<class 'sage.typeset.ascii_art.AsciiArt'>
```

(continues on next page)
build_list \( (l, \text{baseline=0}) \)

Return a character art output of a list.

build_set \( (s, \text{baseline=0}) \)

Return a character art output of a set.

build_tuple \( (t, \text{baseline=0}) \)

Return a character art output of a tuple.

concatenate \( (\text{iterable}, \text{separator}, \text{empty=\text{None}}, \text{baseline=0}, \text{nested=\text{False}}) \)

Concatenate multiple character art instances

The breakpoints are set as the breakpoints of the \text{separator} together with the breakpoints of the objects in \text{iterable}. If there is \text{None}, the end of the separator is used.

INPUT:

- \text{iterable} – iterable of character art
- \text{separator} – character art; the separator in-between the iterable
- \text{empty} – an optional character art which is returned if \text{iterable} is empty
- \text{baseline} – (default: 0) the baseline of the object
- \text{nested} – boolean (default: \text{False}); if True, each of the character art objects is treated as a nested element, so that line breaks at the separator are preferred over line breaks inside the character art objects

EXAMPLES:

```python
sage: i2 = identity_matrix(2)  # _
\rightarrow \text{needs sage.modules}
\text{sage: ascii_art(i2, i2, i2, sep=ascii_art(1/x))}  # _
\rightarrow \text{needs sage.modules sage.symbolic}
\begin{align*}
1 & 1 \\
[1 0] & [1 0] & [1 0] \\
[0 1]x[0 1]x[0 1]
\end{align*}
```

```python
>>> from sage.all import *  
>>> i2 = identity_matrix(Integer(2))  
\rightarrow \text{# needs sage.modules}
>>> ascii_art(i2, i2, i2, sep=ascii_art(Integer(1)/x))  
\rightarrow \text{# needs sage.modules sage.symbolic}
\begin{align*}
1 & 1 \\
[1 0] & [1 0] & [1 0] \\
[0 1]x[0 1]x[0 1]
\end{align*}
```

parse_keywords \( (\text{kwds}) \)

Parse the keyword input given by the dict \text{kwds}.

INPUT:

4.1. Formatted Output
• kwds – a dict

OUTPUT:
A triple:
• the separator
• the baseline
• the baseline of the separator

**Warning:** The input is a dict, not a list of keyword arguments.

**Note:** This will remove `sep`/`separator` and `baseline` from `kwds` if they are specified.

### 4.1.4 ASCII Art

This file contains:
• *AsciiArt* a simple implementation of an ASCII art object,
• *ascii_art()* a function to get the ASCII art representation of any object in Sage,

**AUTHOR:**
• Jean-Baptiste Priez (2013-04): initial version

**EXAMPLES:**

```plaintext
sage: # needs sage.symbolic
sage: n = var('n')
```

```plaintext
sage: integrate(n^2/x, x)
n^2*log(x)
```

```plaintext
sage: ascii_art(integrate(n^2/x, x))
2
n *log(x)
```

```plaintext
sage: ascii_art(integrate(n^2/(pi*x), x))
2
n *log(x)
---------
pi
```

```plaintext
sage: ascii_art(list(Partitions(6)))
#...
```

```plaintext
>>> from sage.all import *
>>> # needs sage.symbolic
>>> n = var('n')
>>> integrate(n**Integer(2)/x, x)
```

(continues on next page)
This method `ascii_art()` could be automatically used by the display hook manager activated by the magic function:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
```

```
sage: shell.run_cell('
from sage.all import *

# Display ascii_art

sage: ascii_art(factorial(i)*x^i, i, 0, 10)

4 + 4*x + 2*x^2 + x^3
```

```
sage: shell.quit()
```

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

```
sage: from sage.all import *

# Display ascii_art

sage: ascii_art(3/(7*x))

3
---
7*x
```

```
sage: shell.quit()
```

```
sage: shell.run_cell('
from sage.all import *

# Display simple

sage: ascii_art(3/(7*x))

3
---
7*x
```

```
sage: shell.quit()
```

4.1. Formatted Output
... from sage.repl.interpreter import get_test_shell
>>> shell = get_test_shell()
>>> shell.run_cell('disp(expr)')
#...

# needs sage.symbolic
>>> shell.run_cell('sum(factorial(i)*x^i, i, 0, 10)')
362880*x + 362880*x + 40320*x + 5040*x + 720*x + 120*x + 24*x + 6*x
+ 2*x + x + 1
>>> shell.run_cell('3/(7*x)')

# needs sage.symbolic
3

---
7*x
>>> shell.run_cell('list(Compositions(5))')
[ [ ], [ * ], [ * * ], [ * * * ], [ * * * * ], [ * * * * * ] ]

# needs sage.combinat

>>> shell.run_cell('disp(simple)')
>>> shell.quit()
class sage.typeset.ascii_art.AsciArt (lines=[], breakpoints=[], baseline=None)

Bases: CharacterArt

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)

EXAMPLES:
sage: i = var('i')  # needs sage.symbolic
sage: ascii_art(sum(pi^i/factorial(i)*x^i, i, 0, oo))  # needs sage.symbolic
pi*x
e
>>> from sage.all import *
>>> i = var('i')  # needs sage.symbolic
>>> ascii_art(sum(pi**i/factorial(i)**x**i, i, Integer(0), oo))  # needs sage.symbolic
pi*x
e
sage.typeset.ascii_art(*obj, **kwds)

Return an ASCII art representation

INPUT:

- *obj – any number of positional arguments, of arbitrary type. The objects whose ascii art representation we want.
- sep – optional 'sep=...' keyword argument (or 'separator'). Anything that can be converted to ascii art (default: empty ascii art). The separator in-between a list of objects. Only used if more than one object given.
- baseline – (default: 0) the baseline for the object
- sep_baseline – (default: 0) the baseline for the separator

OUTPUT:

AsciiArt instance.

EXAMPLES:

sage: result = ascii_art(integral(exp(x+x^2)/(x+1), x))  # needs sage.symbolic

/  
|  
|  2  
| x + x  
e  
| ------- dx  
| x + 1  
/  

>>> from sage.all import *
>>> result = ascii_art(integral(exp(x+x**Integer(2))/(x+Integer(1)), x))  # needs sage.symbolic

/  
|  
|  
|  
| ------- dx  
|  
/  

(continues on next page)
We can specify a separator object:

```python
sage: ident = lambda n: identity_matrix(ZZ, n)
sage: ascii_art(ident(1), ident(2), ident(3), sep=' : ') # needs sage.modules
```

```
\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 1 \\
\end{bmatrix}: \\
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 1 \\
\end{bmatrix}
\]
```

We can specify the baseline:

```python
sage: ascii_art(ident(2), baseline=-1) + ascii_art(ident(3)) # needs sage.modules
```

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

We can determine the baseline of the separator:

```python
sage: ascii_art(ident(1), ident(2), ident(3), sep=' -- ', sep_baseline=-1) # needs sage.modules
```

```
\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]
```
If specified, the `sep_baseline` overrides the baseline of an ascii art separator:

```python
sage: sep_line = ascii_art('\n'.join(' ' for _ in range(6)), baseline=6)
sage: ascii_art(*Partitions(6), separator=sep_line, sep_baseline=0)  # needs sage.combinat sage.libs.flint
```

```
| | | | | | |
| | | | | | *
| | | | | ** |
| | ** | * | ** |
| * | ** | * | ** |
| | | | | | |
```

```python
>>> from sage.all import *
>>> sep_line = ascii_art('\n'.join(' ' for _ in range(Integer(6))), baseline=Integer(6))
```

```python
>>> ascii_art(*Partitions(Integer(6)), separator=sep_line, sep_baseline=Integer(0))  # needs sage.combinat sage.libs.flint
```

```
| | | | | | |
| | | | | | *
| | | | | ** |
| | ** | * | ** |
| * | ** | * | ** |
| | | | | | |
```

### 4.1.5 Unicode Art

This module implements ascii art using unicode characters. It is a strict superset of `asciic_art`.

```python
class sage.typeset.unicode_art.UnicodeArt(lines=[], breakpoints=[], baseline=None)
```

Bases: `CharacterArt`

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)

**EXAMPLES:**

```python
sage: i = var('i')
```

```python
sage: unicode_art(sum(pi^i/factorial(i)*x^i, i, 0, oo))  # needs sage.combinat sage.libs.flint
```

```
π⋅x  
```

```python
>>> from sage.all import *
>>> i = var('i')  
```

```python
>>> unicode_art(sum(pi**i/factorial(i)*x**i, i, Integer(0), oo))  
```

```
π⋅x  
```
sage.typeset.unicode_art.unicode_art(*obj, **kwds)

Return an unicode art representation

**INPUT:**

- *obj – any number of positional arguments, of arbitrary type. The objects whose ascii art representation we want.
- sep – optional 'sep='...’ keyword argument (or 'separator'). Anything that can be converted to unicode art (default: empty unicode art). The separator in-between a list of objects. Only used if more than one object given.
- baseline – (default: 0) the baseline for the object
- sep_baseline – (default: 0) the baseline for the separator

**OUTPUT:**

UnicodeArt instance.

**EXAMPLES:**

```
sage: result = unicode_art(integral(exp(sqrt(x))/(x+pi), x))               # needs sage.symbolic
...
sage: result

sage: ident = lambda n: identity_matrix(ZZ, n)
sage: unicode_art(ident(1), ident(2), ident(3), sep=' : ')                  # needs sage.modules

>>> from sage.all import *
>>> result = unicode_art(integral(exp(sqrt(x))/(x+pi), x))                  # needs sage.symbolic
...
>>> result

>>> ident = lambda n: identity_matrix(ZZ, n)
>>> unicode_art(ident(Integer(1)), ident(Integer(2)), ident(Integer(3)), sep=' : ')  # needs sage.modules

```

If specified, the sep_baseline overrides the baseline of an unicode art separator:
Utilities, Release 10.4

```
sage: sep_line = unicode_art(r'
'.join(' | ' for _ in range(5)), baseline=5)
sage: unicode_art(*AlternatingSignMatrices(3),
                    separator=sep_line, sep_baseline=1)

\[
\begin{array}{cccc}
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
\end{array}
\]

sage.typeset.unicode_art.\texttt{unicode_subscript}(x)

Return the integer \( x \) as a superscript.

EXAMPLES:

```
sage: from sage.typeset.unicode_art import unicode_subscript
sage: unicode_subscript(15123902)
'\text{\textsuperscript{15123902}}'
sage: unicode_subscript(-712)
'\text{\textsuperscript{-712}}'
```

```
>>> from sage.all import *
>>> from sage.typeset.unicode_art import unicode_subscript
>>> unicode_subscript(Integer(15123902))
'\text{\textsuperscript{15123902}}'
>>> unicode_subscript(-Integer(712))
'\text{\textsuperscript{-712}}'
```

sage.typeset.unicode_art.\texttt{unicode_superscript}(x)

Return the rational number \( x \) as a superscript.

EXAMPLES:

```
sage: from sage.typeset.unicode_art import unicode_superscript
sage: unicode_superscript(15123902)
'\textsuperscript{15123902}'
sage: unicode_superscript(-712/5)
'\textsuperscript{-712/5}'
```

```
>>> from sage.all import *
>>> from sage.typeset.unicode_art import unicode_superscript
>>> unicode_superscript(Integer(15123902))
'\textsuperscript{15123902}'
>>> unicode_superscript(-Integer(712)/Integer(5))
'\textsuperscript{-712/5}'
```
4.1.6 Unicode Characters

This module provides Python identifiers for Unicode characters corresponding to various mathematical symbols.

The syntax is `unicode_XXX`, where `XXX` is the LaTeX name of the symbol, stripped of any backslash or any curly brace (e.g. `unicode_mathbbR` for \( \mathbb{R} \)).

**EXAMPLES:**

Operators:

```python
from sage.typeset.unicode_characters import unicode_otimes
sage: unicode_otimes
⊗
sage: hex(ord(_))
0x2297

from sage.typeset.unicode_characters import unicode_bigotimes
sage: unicode_bigotimes
⨂
sage: hex(ord(_))
0x2a02

from sage.typeset.unicode_characters import unicode_wedge
sage: unicode_wedge
∧
sage: hex(ord(_))
0x2227

from sage.typeset.unicode_characters import unicode_bigwedge
sage: unicode_bigwedge
⋀
sage: hex(ord(_))
0x22c0

from sage.typeset.unicode_characters import unicode_partial
sage: unicode_partial
∂
sage: hex(ord(_))
0x2202
```

(continues on next page)
Utilities, Release 10.4

>>> from sage.typeset.unicode_characters import unicode_partial

'∂'

>>> hex(ord(_))
'0x2202'

Arrows:

sage: from sage.typeset.unicode_characters import unicode_to
sage: unicode_to
'→'

sage: hex(ord(_))
'0x2192'

sage: from sage.typeset.unicode_characters import unicode_mapsto
sage: unicode_mapsto
'↦'

sage: hex(ord(_))
'0x21a6'

Letters:

sage: from sage.typeset.unicode_characters import unicode_mathbbR
sage: unicode_mathbbR
'ℝ'

sage: hex(ord(_))
'0x211d'

sage: from sage.typeset.unicode_characters import unicode_mathbbC
sage: unicode_mathbbC
'ℂ'

sage: hex(ord(_))
'0x2102'

>>> from sage.all import *

>>> from sage.typeset.unicode_characters import unicode_to

'→'

>>> hex(ord(_))
'0x2192'

>>> from sage.typeset.unicode_characters import unicode_mapsto

'↦'

>>> hex(ord(_))
'0x21a6'

>>> from sage.all import *

>>> from sage.typeset.unicode_characters import unicode_mathbbR

'ℝ'

>>> hex(ord(_))
'0x211d'

>>> from sage.typeset.unicode_characters import unicode_mathbbC

'ℂ'

>>> hex(ord(_))
'0x2102'

Chapter 4. Formatted Output
4.1.7 Repr formatting support

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

String representing coefficients in a linear combination.

**INPUT:**
- `c` – a coefficient (i.e., an element of a ring)

**OUTPUT:**
A string

**EXAMPLES:**

```python
sage: from sage.misc.repr import coeff_repr
sage: coeff_repr(QQ(1/2))
'1/2'
sage: coeff_repr(-x^2)  # needs sage.symbolic
'(-x^2)'
sage: coeff_repr(QQ(1/2), is_latex=True)
'\frac{1}{2}'
sage: coeff_repr(-x^2, is_latex=True)  # needs sage.symbolic
'\left(-x^{2}\right)'
```

```python
>>> from sage.all import *
>>> from sage.misc.repr import coeff_repr
>>> coeff_repr(QQ(Integer(1)/Integer(2)))
'1/2'
>>> coeff_repr(-x**Integer(2))  # needs sage.symbolic
'(-x^2)'
>>> coeff_repr(QQ(Integer(1)/Integer(2)), is_latex=True)
'\frac{1}{2}'
>>> coeff_repr(-x**Integer(2), is_latex=True)  # needs sage.symbolic
'\left(-x^{2}\right)'
```

`sage.misc.repr.repr_lincomb(terms, 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: a space if scalar_mult is `'*'`; otherwise scalar_mult)
- `coeffs` – for backward compatibility

**OUTPUT:**
- `str` – a string

**EXAMPLES:**
Utilities, Release 10.4

```
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)
```

```
2**a - 3**c

```python
>>> repr_lincomb([[`a`,-Integer(1)], [`b`,Integer(2)], [`c`,Integer(3)]], scalar_mult='**')
'-a + 2**b + 3**c'
```

Examples for `scalar_mult` and `is_latex`:

```python
sage: repr_lincomb([[`a`,-1], [`b`,2], [`c`,3]], is_latex=True)
'-a + 2 b + 3 c'
sage: repr_lincomb([[`a`,-1], [`b`,-1], [`c`,3]], is_latex=True, scalar_mult='*')
'-a - b + 3 c'
sage: repr_lincomb([[`a`,-2], [`b`,2], [`c`,-3]], is_latex=True, scalar_mult='**')
'-2**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'
sage: repr_lincomb([[`a`,-2], [`b`,-1], [`c`,-3]], is_latex=True, latex_scalar_mult='')
'-2a - b - 3c'
```

```python
>>> from sage.all import *
```
4.1.8 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: sage_input(3)
3
sage: sage_input((polygen(RR) + RR(pi))^2, verify=True)  # Needs sage.symbolic
# 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: sage_input(GF(2)(1), verify=True)
# Verified
GF(2)(1)
```

(continues on next page)
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.:

```python
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.:

```python
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.):

```
>>> from sage.all import *
>>> def qq_sage_input_v1(self, sib, coerced):
...     return sib(self.numerator())/sib(self.denominator())
```

```
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]
```

```
>>> from sage.misc.sage_input import SageInputBuilder
>>> def qq_sage_input_v2(self, sib, coerced):
...     return sib(self.numerator())/sib.int(self.denominator())
```

```
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]
```

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, \(-22(5)/7 \equiv -22(5)/22(7)\).

```
sage: def qq_sage_input_v2(self, sib, coerced):
...     return sib(self.numerator())/sib.int(self.denominator())
```

```
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder
>>> def test_qq_formatter(fmt):
...     results = []
...     for v in [-Integer(5)/Integer(7), QQ(Integer(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
```

```
>>> 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]
```

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` or an `int`).: 

```
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 us get rid of the divisions by 1. These are more complicated, since if we are not careful we will get results in \( \mathbb{Z} \) instead of \( \mathbb{Q} \):

```python
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 \( \text{sage constant or function} \):

```python
sage: test_qq_formatter(qq_sage_input_v3)
[-2Z(5)/7, -2Z(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 \text{sage_input}(\text{polygen}(\mathbb{Q}) - 2/3) would produce \( x + (-2/3) \); if we change to the following code, then we would get \( x - 2/3 \) instead.: 

```python
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
```

```python
sage: test_qq_formatter(qq_sage_input_v4)
[-2Z(5)/7, -2Z(5)/7, -5/7, -5/7, QQ(3), 3, QQ(3), 3]
```
```python
>>> from sage.all import *
>>> def qq_sage_input_v4(self, sib, coerced):
...     num = self.numerator()
...     neg = (num < Integer(0))
...     if neg: num = -num
...     if self.denominator() == Integer(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

>>> 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

```python
class sage.misc.sage_input.SIE_assign(sib, lhs, rhs)
    Bases: SageInputExpression
    This class represents an assignment command.

EXAMPLES:
```sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
    sib.assign(sib.name('foo').x, sib.name('pi'))
{assign: {getattr: {atomic:foo}.x} {atomic:pi}}
```

```python
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder

sib = SageInputBuilder()
    sib.assign(sib.name('foo').x, sib.name('pi'))
{assign: {getattr: {atomic:foo}.x} {atomic:pi}}
```

```python
class sage.misc.sage_input.SIE_binary(sib, op, lhs, rhs)
    Bases: 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()
    sib(3)+5
{binop:+ {atomic:3} {atomic:5}}
```
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder

>>> sib = SageInputBuilder()
>>> sib(Integer(3))+Integer(5)
{binop:+ {atomic:3} {atomic:5}}

class sage.misc.sage_input.SIE_call(sib, func, args, kwargs)

Bases: 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})}

>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder

>>> sib = SageInputBuilder()
>>> sie = sib.name('GF')
>>> sie(Integer(49))
{call: {atomic:GF}({atomic:49})}

class sage.misc.sage_input.SIE_dict(sib, entries)

Bases: 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))})
# → needs sage.symbolic
{dict: {{atomic:TeX}:{call: {atomic:RR}({atomic:3.1415926535897931})},
    {atomic:Metafont}:{call: {atomic:RR}({atomic:2.7182818284590451})}}}

sage: sib.dict({-40:-40, 0:32, 100:212})
{dict: {{unop:- {atomic:40}}:{unop:- {atomic:40}},
    {atomic:0}:{atomic:32}, {atomic:100}:{atomic:212}}}

(continues on next page)
class sage.misc.sage_input.SIE_gen(sib, parent, name)

   Bases: 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',)}}}

>> from sage.all import *
>> from sage.misc.sage_input import SageInputBuilder
>> sib = SageInputBuilder()
>> 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: SageInputExpression

   This class represents an expression that can create a sage parent with named generators, optionally using the sage
   preparsers 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',)}}

>> from sage.all import *
>> from sage.misc.sage_input import SageInputBuilder
>> sib = SageInputBuilder()
>> qq = sib.name('QQ')
>> 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: SageInputExpression

   This class represents a getattr node in a sage_input() expression tree.

   EXAMPLES:
```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: sie = sib.name('CC').gen()
sage: sie
{call: {getattr: {atomic:CC}.gen}()}
```

```python
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder

>>> sib = SageInputBuilder()
>>> sie = sib.name('CC').gen()
>>> sie
{call: {getattr: {atomic:CC}.gen}()}
```

```python
class sage.misc.sage_input.SIE_import_name(sib, module, name, alt_name=None)

Bases: SageInputExpression

This class represents a name which has been imported from a module.

EXAMPLES:

```python
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}
```

```python
class sage.misc.sage_input.SIE_literal(sib)

Bases: SageInputExpression

An abstract base class for literals (basically, values which consist of a single token).

EXAMPLES:

```python
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
```

```python
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder, SIE_literal
```

(continues on next page)
class sage.misc.sage_input.SIELiteralStringrep(sib, n)

Bases: SIELiteral

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:**

```python
class sage.misc.sage_input.SIELiteralStringrep(sib, n)

Bases: SIELiteral

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:**

```python
class sage.misc.sage_input.SIELiteralStringrep(sib, n)

Bases: SIELiteral

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:**

```python
class sage.misc.sage_input.SIELiteralStringrep(sib, n)

Bases: SIELiteral

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:**

```python
class sage.misc.sage_input.SIELiteralStringrep(sib, n)

Bases: SIELiteral

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:**

```python
class sage.misc.sage_input.SIE_tuple(sib, values, is_list)

Bases: SageInputExpression

This class represents a tuple or list node in a `sage_input()` expression tree.

EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
>>> sib((1, 'howdy'))
{tuple: ({atomic:1}, {atomic:'howdy'})}
```

```python
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder
```  
```python
>>> sib(Integer(256))
{unop:- {atomic:256}}
```  
```

class sage.misc.sage_input.SIE_unary(sib, op, operand)

Bases: SageInputExpression

This class represents an arithmetic expression with a unary operator and its argument, in a `sage_input()` expression tree.

EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
>>> -sib(256)
{unop:- {atomic:256}}
```  
```python
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder
```  
```python
>>> -sib(Integer(256))
{unop:- {atomic:256}}
```
class sage.misc.sage_input.SageInputAnswer(cmds, expr, locals=None)

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:

```python
sage: from sage.misc.sage_input import SageInputAnswer
sage: v = SageInputAnswer('x = 22
x/7'); v
x = 22
x/7
sage: isinstance(v, tuple)
True
sage: v[0]
x = 22
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>}
```

```python
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputAnswer
>>> v = SageInputAnswer('x = 22
x/7'); v
x = 22
x/7
>>> isinstance(v, tuple)
True
>>> v[Integer(0)]
x = 22

>>> v[Integer(1)]
x/7
>>> len(v)
2
>>> v = SageInputAnswer('', 'sin(3.14)', {'sin': math.sin}); v
LOCALS:
sin: <built-in function sin>
sin(3.14)
>>> v[Integer(0)]
'

>>> v[Integer(1)]
'sin(3.14)'
>>> v[Integer(2)]
{'sin': <built-in function sin>}
```

class sage.misc.sage_input.SageInputBuilder(allow_locals=False, preparse=True)

Bases: object

An instance of this class is passed to `_sage_input_` methods. It keeps track of the current state of the
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:**

```python
sage: from sage.misc.sage_input import SageInputBuilder

>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder

We can create a `SageInputBuilder`, use it to create some `SageInputExpression`es, and get a result. (As mentioned above, this is only useful for testing or doctesting; normally you would just use `sage_input()`):

```python
sage: sib = SageInputBuilder()
sage: sib.result((sib(3) + sib(4)) * (sib(5) + sib(6)))
(3 + 4)*(5 + 6)
```  

```python
>>> from sage.all import *

```
• x – an arbitrary value
• sie – a \texttt{SageInputExpression}
• name – a requested variable name

Enters x and sie in a cache, so that subsequent calls \texttt{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:**

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
```

```python
sage: sib(GF(101)(42), sie42, 'the_ultimate_answer')
sage: sib.result(sib(GF(101)(42)) + sib(GF(101)(42)))
```

```python
the_ultimate_answer = GF(101)(42)
the_ultimate_answer + the_ultimate_answer
```

```python
>>> from sage.all import *
```

```python
>>> sib = SageInputBuilder()
```

```python
>>> sib(GF(Integer(101))(Integer(42)), sie42, 'the_ultimate_answer')
```

```python
>>> sib.result(sib(GF(Integer(101))(Integer(42))) + sib(GF(Integer(101))(Integer(43))))
```

```python
GF_101 = GF(101)
GF_101(42) + GF_101(43)
```

Note that we don’t assign the result to a variable if the value is only used once.:

```python
sage: sib = SageInputBuilder()
```

```python
sage: sib(GF(101)(42), sie42, 'the_ultimate_answer')
```

```python
sage: sib.result(sib(GF(101)(42)) + sib(GF(101)(43)))
```

```python
GF_101 = GF(101)
GF_101(42) + GF_101(43)
```

**command** \( (v, \text{cmd}) \)

**INPUT:**

• v, cmd – \texttt{SageInputExpression}

Attaches a command to v, which will be executed before v is used. Multiple commands will be executed in the order added.
EXAMPLES:

```python
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]))
```

```python
si = []
si.append(1)
si.extend([2, 3])
```

```python
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder

>>> sib = SageInputBuilder()
>>> incr_list = sib([])
>>> sib.command(incr_list, incr_list.append(Integer(1)))
>>> sib.command(incr_list, incr_list.extend([Integer(2), Integer(3)]))
```

```python
si = []
si.append(1)
si.extend([2, 3])
```

dict (entries)

Given a dictionary, or a list of (key, value) pairs, produces a `SageInputExpression` representing the dictionary.

EXAMPLES:

```python
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}
```

```python
sage: sib.result(sib.dict([(hello, sunshine), (goodbye, rain)]))
{hello:sunshine, goodbye:rain}
```

```python
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder

>>> sib = SageInputBuilder()
>>> sib.result(sib.dict({Integer(1):Integer(1), Integer(2):Integer(5)/Integer(2), Integer(3):Integer(100)/Integer(3)}))
{1:1, 2:5/2, 3:100/3}
```

```python
>>> 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:
Utilities, Release 10.4

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: sib.result(sib.empty_subscript(sib(2) + sib(3)))
(2 + 3)[]
```

```python
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder
>>> sib = SageInputBuilder()
>>> sib.result(sib.empty_subscript(sib(Integer(2)) + sib(Integer(3))))
(2 + 3)[]
```

The following calls this method indirectly:

```python
sage: sage_input(polygen(ZZ['y']))
R.<x> = ZZ['y'][]
x
```

```python
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder
>>> sib = SageInputBuilder()
>>> sib.result(sib.float_str(repr(RR(e))))
2.71828182845905
```

```python
float_str(n)
```

Given a string representing a floating-point number, produces a `SageInputExpression` that formats as that string.

EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: sib.result(sib.float_str(repr(RR(e))))     # needs sage.symbolic
2.71828182845905
```

```python
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder
>>> sib = SageInputBuilder()
>>> sib.result(sib.float_str(repr(RR(e))))     # needs sage.symbolic
2.71828182845905
```

```python
gen(parent, n=0)
```

Given a parent, returns a `SageInputExpression` for the \(n\)-th (default 0) generator of the parent.

EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: sib.result(sib.gen(ZZ['y']))
R.<y> = ZZ['y'][]
y
```

424 Chapter 4. Formatted Output
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:

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:
Utilities, Release 10.4

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:

```python
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder

>>> x = polygen(ZZ)
>>> my_42 = Integer(42)*x
>>> sie42 = sib(my_42)
>>> sib.id_cache(my_42, sie42, 'the_ultimate_answer')
>>> 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:

```python
>>> 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:

```python
>>> my_42 = 42*x
>>> sie42 = sib(my_42)
>>> sib.id_cache(my_42, sie42, the_ultimate_answer)
>>> sib.result(sib(my_42) + sib(43*x))
R.<x> = ZZ[]
42*x + 43*x
```

```python
>>> sib = SageInputBuilder()
>>> my_42 = Integer(42)*x
>>> sie42 = sib(my_42)
>>> sib.id_cache(my_42, sie42, 'the_ultimate_answer')
>>> sib.result(sib(my_42) + sib(Integer(43)*x))
R.<x> = ZZ[]
42*x + 43*x
```
```python
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:

```python
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
```

```python
>>> from sage.all import *

>>> sib = SageInputBuilder()

>>> v1 = sib.import_name('sage.foo', 'poly')

>>> v2 = sib.import_name('sage.bar', 'poly')

>>> 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 or a Python int, depending on its size and whether the preparser is enabled.

**INPUT:**

- `n` – a Sage Integer or a Python int

**EXAMPLES:**

```python
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.result(sib.int(-3^50))
-717897987691852588770249
```

(continues on next page)
name \((n)\)

Given a string representing a Python name, produces a \texttt{SageInputExpression} for that name.

**EXAMPLES:**

```python
sage: from sage.all import *
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 \texttt{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 \texttt{name} parameter.

**EXAMPLES:**

```python
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
```
Utilities, Release 10.4

4.1. Formatted Output

```python
...:    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()  # doctest: +NORMALIZE_WHITESPACE
sage: sib.result(par_sie)
make_a_parent(names=('foo', 'bar'))

sage: sib, par_sie = test_setup()  # doctest: +NORMALIZE_WHITESPACE
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)  # doctest: +NORMALIZE_WHITESPACE
sage: sib.result(par_sie)
make_a_parent(names=('foo', 'bar'))

sage: sib, par_sie = test_setup(preparse=False)  # doctest: +NORMALIZE_WHITESPACE
sage: sib.result(sib(3) * sib.gen("some parent", 0))
the_thing = make_a_parent(names=('foo', 'bar'))
foo,bar = the_thing.gens()
2*3*foo

sage: sib, par_sie = test_setup(use_gens=False)  # doctest: +NORMALIZE_WHITESPACE
sage: sib.result(par_sie)
make_a_parent(names=('foo', 'bar'))

sage: sib, par_sie = test_setup(use_gens=False)  # doctest: +NORMALIZE_WHITESPACE
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()  # doctest: +NORMALIZE_WHITESPACE
sage: sib.result(par_sie - sib.gen("some parent", 1))
the_thing.<foo,bar> = make_a_parent()
the_thing - bar

>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder

>>> 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,
```

(continues on next page)
... return sib, result

gens_syntax=gens_sie)

>>> sib, par_sie = test_setup()
>>> sib.result(par_sie)
make_a_parent(names=('foo', 'bar'))

the_thing.<foo,bar> = make_a_parent()
3*foo

>>> sib, par_sie = test_setup(preparse=False)
>>> sib.result(par_sie)
make_a_parent(names=('foo', 'bar'))

the_thing = make_a_parent(names=('foo', 'bar'))
foo, bar = the_thing.gens()
ZZ(3)*foo

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
```

```
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder
>>> SageInputBuilder().preparse()
```
True

```python
>>> 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:**

```python
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
```

```
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder
```

```python
>>> sib = SageInputBuilder()
>>> sib.result(sib.prod([-Integer(1), Integer(0), Integer(1), -Integer(2)]))
-1*0*1*-2
>>> sib = SageInputBuilder()
>>> sib.result(sib.prod([-Integer(1), Integer(0), Integer(1), Integer(2)],
... simplify=True))
0
>>> sib = SageInputBuilder()
>>> sib.result(sib.prod([-Integer(1), Integer(2), -Integer(3), -Integer(4)],
... simplify=True))
-2*3*4
>>> sib = SageInputBuilder()
>>> sib.result(sib.prod([-Integer(1), Integer(1), -Integer(1), -Integer(1)],
... simplify=True))
```
Utilities, Release 10.4

>>> sib = SageInputBuilder()
>>> sib.result(sib.prod([Integer(1), Integer(1), Integer(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()
>>> r = sib.result(sib(6) * sib(7)); r
6*7
>>> 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:

Without explicitly using .share(), string literals are not shared:

sage: from sage.misc.sage_input import SageInputBuilder
>>> sib = SageInputBuilder()
>>> e = sib('hello')
>>> sib.result(sib((e, e)))
('hello', 'hello')

See the difference if we use .share():

sage: sib = SageInputBuilder()
>>> e = sib('hello')
>>> sib.result(sib((e, e)).share())
('hello', 'hello')
sage: sib = SageInputBuilder()
sage: e = sib('hello')
sage: sib.share(e)
sage: sib.result(sib((e, e)))

si = 'hello'
(si, si)

```python
>>> from sage.all import *
>>> sib = SageInputBuilder()
>>> e = sib('hello')
>>> sib.share(e)
>>> 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:**

```python
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:

```python
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()
```

```python
>>> from sage.all import *
>>> from sage.misc.sage_input import SageInputBuilder

>>> sib = SageInputBuilder()
>>> e = sib.name('MatrixSpace')(ZZ, Integer(10), Integer(10))
>>> sib.use_variable(e, 'MS')
>>> sib.result(e.zero_matrix())
MS = MatrixSpace(ZZ, 10, 10)
MS.zero_matrix()
```

Without the call to use_variable, we get this instead:

```python
sage: sib = SageInputBuilder()
sage: e = sib.name('MatrixSpace')(ZZ, 10, 10)
sage: sib.result(e.zero_matrix())
MatrixSpace(ZZ, 10, 10).zero_matrix()
```

```python
>>> from sage.all import *

>>> sib = SageInputBuilder()
>>> e = sib.name('MatrixSpace')(ZZ, Integer(10), Integer(10))
>>> 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:

```python
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)
```

```python
>>> from sage.all import *

>>> sib = SageInputBuilder()
>>> e = sib.name('MatrixSpace')(ZZ, Integer(10), Integer(10))
>>> 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 \texttt{SageInputExpression} objects construct a new \texttt{SageInputExpression} representing that operation. That is, if \( a \) is a \texttt{SageInputExpression}, then \( a + b \) constructs a \texttt{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 \texttt{SageInputBuilder} and \texttt{SageInputExpression}.

Values of type \texttt{SageInputExpression} print in a fairly ugly way, that reveals the internal structure of the expression tree.

\begin{verbatim}
class sage.misc.sage_input.SageInputFormatter
    Bases: object

    An instance of this class is used to keep track of variable names and a sequence of generated commands during the \texttt{sage_input()} formatting process.

    \texttt{format}(e, \texttt{prec})

        Format a Sage input expression into a string.

        INPUT:

        \begin{itemize}
        \item \( e \) – a \texttt{SageInputExpression}
        \item \texttt{prec} – an integer representing a precedence level
        \end{itemize}

        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\(_\_\texttt{sie_format}\_\) 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(Integer(5)))

        sage: from sage.all import *
        sage: from sage.misc.sage_input import SageInputBuilder, SageInputFormatter
        >>> sib = SageInputBuilder()
        >>> sif = SageInputFormatter()
        >>> sie = sib(GF(5))

        Here we cheat by calling method\(_\_\texttt{sie_prepare}\_\) 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)

        \end{verbatim}

\end{verbatim}
We demonstrate the use of commands, by showing how to construct code that will produce a random matrix:

```python
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
'si = matrix(ZZ, 10, 10)\nsi.randomize()\n'
```

```python
>>> from sage.all import *
>>> sib = SageInputBuilder()
>>> sif = SageInputFormatter()
>>> sie = sib.name('matrix')(sib.name('ZZ'), 10, 10)
>>> sib.command(sie, sie.randomize())
>>> sie._sie_prepare(sif)
>>> sif._commands
'GF_5\n'
```

```python
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'
```

**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:**
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:

```python
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._names, sif._dup_names
({'x'}, {})  # 'x' is unique
sage: sif.register_name('y')
sage: sif._names, sif._dup_names
({'x', 'y'}, {})  # 'y' is unique
sage: sif.register_name('x')
sage: sif._names, sif._dup_names
({'x', 'y'}, {'x': 0})  # 'x' is now duplicated
```

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 True) 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:**

```python
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))
```

```python
>>> from sage.all import *

>>> sage_input(GF(Integer(2))(Integer(0)), GF(Integer(2))(Integer(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:

```python
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
```

```python
>>> from sage.all import *

>>> K = GF(Integer(5))['x']; (x,) = K._first_ngens(1)

>>> sage_input(x**Integer(3) + Integer(2)*x, verify=True)
# Verified
R.<x> = GF(5)[]
x^3 + 2*x
```

The result of `sage_input()` is actually a pair of strings with a special `__repr__` method to print nicely:

```python
sage: # needs sage.rings.real_mpfr sage.symbolic
sage: r = sage_input(RealField(20)(pi), verify=True)
sage: r
# Verified
RealField(20)(3.1415939753589793)
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

```python
sage: isinstance(r, tuple)
True
sage: len(r)
2
sage: tuple(r)
(('# Verified\n', 'RealField(20)(3.1415939)'))
```

```python
>>> from sage.all import *
>>> # needs sage.rings.real_mpfr sage.symbolic
>>> r = sage_input(RealField(Integer(20))(pi), verify=True)
>>> r
# Verified
RealField(20)(3.1415939)
>>> isinstance(r, tuple)
True
>>> len(r)
2
>>> tuple(r)
('(# Verified\n', 'RealField(20)(3.1415939)')
```

We cannot find an input form for a function:

```python
sage: sage_input((3, lambda x: x))
Traceback (most recent call last):
... ValueError: cannot convert <function <lambda> at 0x...> to sage_input form
```

```python
>>> from sage.all import *
>>> sage_input((Integer(3), lambda x: x))
Traceback (most recent call last):
... ValueError: cannot 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`:

```python
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...'}}>)
```

```python
>>> from sage.all import *
>>> r = sage_input((Integer(3), lambda x: x), verify=True, allow_locals=True)
>>> r
LOCALS:
  _sil1: <function <lambda> at 0x...>
# Verified
(3, _sil1)
>>> tuple(r)
('(# Verified\n', '(3, _sil1)', '{'_sil1': <function <lambda> at 0x...'}}>)
```

`sage.misc.sage_input.verify_same(a, b)`

4.1. Formatted Output
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(1r, 1)
Traceback (most recent call last):
  ...  
AssertionError: 'int' object has no attribute 'parent'
  ...  
AssertionError
sage: verify_same(1r, 1)
Traceback (most recent call last):
  ...  
AssertionError
sage: verify_same(1r, 1)
Traceback (most recent call last):
  ...  
AssertionError
```

```
>>> from sage.all import *
>>> from sage.misc.sage_input import verify_same

>>> verify_same(1, 1)
>>> verify_same(1, 2)
Traceback (most recent call last):
  ...  
AssertionError: Expected 1 == 2
>>> verify_same(1, 1)
Traceback (most recent call last):
  ...  
AssertionError
>>> verify_same(1, 1)
Traceback (most recent call last):
  ...  
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)
```

```
>>> from sage.all import *
>>> from sage.misc.sage_input import verify_si_answer
>>> verify_si_answer(Integer(1), '1', True)
>>> verify_si_answer(Integer(1), '1', False)
Traceback (most recent call last):
  ...  
AttributeError: 'int' object has no attribute 'parent'
>>> verify_si_answer(Integer(1), ZZ(1), None)
```

### 4.1.9 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)

```python
class sage.misc.table.table(rows=None, columns=None, header_row=False, header_column=False, frame=False, align='left')
```

**Bases:** `SageObject`

Display a rectangular array as a table, either in plain text, LaTeX, or html.

**INPUT:**

• `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
```

(continues on next page)
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”.

```python
sage: table(rows, header_row=True)
\begin{tabular}{|l|l|l|}
\hline
| a | b | c |
\hline
| 100 | 2 | 3 |
\hline
| 4 | 5 | 60 |
\hline
\end{tabular}
```

(continues on next page)
<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>60</th>
</tr>
</thead>
</table>

```latex
\begin{tabular}{|l|l|l|} 
\hline
a & b & c \\
\hline
$100$ & $2$ & $3$ \\
$4$ & $5$ & $60$ \\
\hline
\end{tabular}
```

```python
sage: table(rows, header_row=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=["$x$", r"\$\sin(x)\$"], frame=True)
```

```
<table>
<thead>
<tr>
<th>$x$</th>
<th>$\sin(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>0.84</td>
</tr>
<tr>
<td>2</td>
<td>0.91</td>
</tr>
<tr>
<td>3</td>
<td>0.14</td>
</tr>
</tbody>
</table>
```

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=['$x$', r'$\sin(x)$'], frame=True)
```

```
<table>
<thead>
<tr>
<th>$x$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sin(x)$</td>
<td>0.00</td>
<td>0.84</td>
<td>0.91</td>
<td>0.14</td>
</tr>
</tbody>
</table>
```

(continues on next page)
or by passing the original data as the columns of the table and using header_column instead of header_row:

```python
sage: table(columns=[(x, n(sin(x), digits=2)) for x in [0..3]],
         header_column=['$x$', r'$\sin(x)$'], frame=True)
```

<table>
<thead>
<tr>
<th>$x$</th>
<th>$\sin(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>0.84</td>
</tr>
<tr>
<td>2</td>
<td>0.91</td>
</tr>
<tr>
<td>3</td>
<td>0.14</td>
</tr>
</tbody>
</table>

or by taking the `transpose()` of the original table:

```python
sage: table(rows=[(x, n(sin(x), digits=2)) for x in [0..3]],
         header_row=['$x$', r'$\sin(x)$'], frame=True).transpose()
```

<table>
<thead>
<tr>
<th>$x$</th>
<th>$\sin(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>0.84</td>
</tr>
<tr>
<td>2</td>
<td>0.91</td>
</tr>
<tr>
<td>3</td>
<td>0.14</td>
</tr>
</tbody>
</table>

In either plain text or LaTeX, entries in tables can be aligned to the left (default), center, or right:

```python
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
```

(continues on next page)
To generate HTML you should use `html(table(...))`:

```python
sage: # needs sage.symbolic
sage: data = [['$x$', r'$\sin(x)$'] + [(x, n(sin(x), digits=2))
....:     for x in [0..3]]
sage: output = html(table(data, header_row=True, frame=True))
sage: type(output)
<class 'sage.misc.html.HtmlFragment'>
sage: print(output)
<div class="notruncate">
  <table border="1" class="table_form">
    <thead>
      <tr>
        <th style="text-align:left">$x$</th>
        <th style="text-align:left">$\sin(x)$</th>
      </tr>
    </thead>
    <tbody>
      <tr class="row-a">
        <td style="text-align:left">0</td>
        <td style="text-align:left">0.00</td>
      </tr>
      <tr class="row-b">
        <td style="text-align:left">1</td>
        <td style="text-align:left">0.84</td>
      </tr>
      <tr class="row-a">
        <td style="text-align:left">2</td>
        <td style="text-align:left">0.91</td>
      </tr>
      <tr class="row-b">
        <td style="text-align:left">3</td>
        <td style="text-align:left">0.91</td>
      </tr>
    </tbody>
  </table>
</div>
```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> data = [['x', r'\sin(x)']] + [(x, n(sin(x), digits=Integer(2)))
...     for x in (ellipsis_range(Integer(0),
...     Ellipsis, Integer(3)))]
>>> output = html(table(data, header_row=True, frame=True))
>>> type(output)
<class sage.misc.html.HtmlFragment>
>>> print(output)
<div class="notruncate">
<table border="1" class="table_form">
<tbody>
<tr>
<th style="text-align:left">x</th>
<th style="text-align:left">\sin(x)</th>
</tr>
<tr class="row-a">
<td style="text-align:left">0</td>
<td style="text-align:left">0.00</td>
</tr>
<tr class="row-b">
<td style="text-align:left">1</td>
<td style="text-align:left">0.84</td>
</tr>
<tr class="row-a">
<td style="text-align:left">2</td>
<td style="text-align:left">0.91</td>
</tr>
<tr class="row-b">
<td style="text-align:left">3</td>
<td style="text-align:left">0.14</td>
</tr>
</tbody>
</table>
</div>
It is an error to specify both rows and columns:

```sage```
```plaintext
table(rows=[[1,2,3], [4,5,6]], columns=[[0,0,0], [0,0,1024]])
```
```
Traceback (most recent call last):
...
ValueError: do not set both 'rows' and 'columns' when defining a table
```
```sage```
```plaintext
table(columns=[[0,0,0], [0,0,1024]])
```
```plaintext
0 0
type 0 1024
```
>>> from sage.all import *
>>> table(rows=[[Integer(1),Integer(2),Integer(3)], [Integer(4),Integer(5),
  →Integer(6)]], columns=[[Integer(0),Integer(0),Integer(0)], [Integer(0),
  →Integer(0),Integer(1024)]])
Traceback (most recent call last):
  ...
ValueError: do not set both 'rows' and 'columns' when defining a table

>>> table(columns=[[Integer(0),Integer(0),Integer(0)], [Integer(0),Integer(0),
  →Integer(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
```

```
>> from sage.all import *
>> table([[Integer(1),Integer(2),Integer(3),Integer(7),Integer(12)], [Integer(4),
  →Integer(5)]]))
1 2
4 5

>> from sage.all import *
>> table([[Integer(1),Integer(2),Integer(3),Integer(7),Integer(12)], [Integer(4),
  →Integer(5)]]))
1 2
4 5
```

```
rich_repr_(display_manager, **kwds)

Rich Output Magic Method

See sage.repl.rich_output for details.

EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
gm = get_display_manager()
sage: t = table([1, 2, 3])
sage: t._rich_repr_(dm)  # the doctest backend does not support html
```
options(**kwds)

With no arguments, return the dictionary of options for this table. With arguments, modify options.

INPUT:

- 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:

```python
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 highlighted. 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.

```python
sage: T = table([[[1,2,3], [4,5,6]], header_row=['a', 'b', 'c'], frame=True])
sage: T
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
sage: T.options(header_row=False)
sage: T
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

```python
>>> from sage.all import *

>>> T = table([[Integer(1), Integer(2), Integer(3)], [Integer(4), Integer(5), Integer(6)]], header_row=['a', 'b', 'c'], frame=True)

>>> T

+---+---+---+
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+

>>> T.options(header_row=False)

>>> T

+---+---+---+
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
+---+---+---+
| 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.
```

```python
>>> from sage.all import *

>>> 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()
+---+---+---+
<table>
<thead>
<tr>
<th>x</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
```

(continues on next page)
>>> from sage.all import *
>>> T = table([[Integer(1),Integer(2),Integer(3)], [Integer(4),Integer(5),
Integer(6)]])
>>> T.transpose()
    1  4
    2  5
    3  6
>>> T = table([[Integer(1),Integer(2),Integer(3)], [Integer(4),Integer(5),
Integer(6)]], header_row=['x', 'y', 'z'], frame=True)
>>> T.transpose()
+---+---+---+
| x | 1 | 4 |
| y | 2 | 5 |
| z | 3 | 6 |
+---+---+---+

### 4.2 HTML and MathML

#### 4.2.1 HTML Fragments

This module defines a HTML fragment class, which holds a piece of HTML. This is primarily used in browser-based notebooks, though it might be useful for creating static pages as well.

This module defines MathJax, an object of which performs the task of producing an HTML representation of any object. The produced HTML is renderable in a browser-based notebook with the help of MathJax.

```python
class sage.misc.html.HTMLFragmentFactory
    Bases: SageObject

eval(s, locals=None)
    Evaluate embedded <sage> tags

    INPUT:

    • s – string.
    • globals – dictionary. The global variables when evaluating s. Default: the current global variables.

    OUTPUT:

    A HtmlFragment instance.

    EXAMPLES:

    sage: a = 123
    sage: html.eval('\langle{sage}a\langle{/sage}⟩')
    '123'
    sage: html.eval('\langle{sage}a\langle{/sage}⟩', locals={'a': 456})
    '456'
```
Utilities, Release 10.4

```python
>>> from sage.all import *
>>> a = Integer(123)
>>> html.eval('<sage>a</sage>')
\(123\)
>>> html.eval('<sage>a</sage>', locals={'a': Integer(456)})
\(456\)
```

**iframe** *(url, height=400, width=800)*

Generate an iframe HTML fragment

**INPUT:**

- **url** – string. A url, either with or without URI scheme (defaults to “http”), or an absolute file path.
  - **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:**

A *HtmlFragment* instance.

**EXAMPLES:**

```python
sage: pretty_print(html.iframe("sagemath.org"))
<iframe height="400" width="800"
src="http://sagemath.org"></iframe>
sage: pretty_print(html.iframe("http://sagemath.org",30,40))
<iframe height="30" width="40"
src="http://sagemath.org"></iframe>
sage: pretty_print(html.iframe("https://sagemath.org",30))
<iframe height="30" width="800"
src="https://sagemath.org"></iframe>
sage: pretty_print(html.iframe("/home/admin/0/data/filename"))
<iframe height="400" width="800"
src="/home/admin/0/data/filename"></iframe>
sage: pretty_print(html.iframe('data:image/png;base64,' +
"iVBORw0KGgoAAAANSUhEUgAAA...: 'AUA...: 'AO9TXL0Y40HwAAAAABJRU5ErkJggg="'))
<iframe height="400" width="800"
src="data:image/png;base64," +
"iVBORw0KGgoAAAANSUhEUgAAAAUAAAAFCAYAAACNbyblAAAAHElEQVQI12P4//8/w38GIAXDIBKE0DHxgljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg=""></iframe>
```

(continues on next page)
class sage.misc.html.HtmlFragment

Bases: str, SageObject

A HTML fragment.

This is a piece of HTML, usually not a complete document. For example, just a `<div>...</div>` piece and not the entire `<html>...</html>`.

EXAMPLES:

```
sage: from sage.misc.html import HtmlFragment
given_string = HtmlFragment('<b>test</b>')

print(given_string)
```

```
<b>test</b>
```

```python
>>> from sage.all import *
>>> from sage.misc.html import HtmlFragment
>>> given_string = HtmlFragment('<b>test</b>')
```

```python
<test>
```

```
>>> from sage.all import *
>>> from sage.misc.html import HtmlFragment
>>> given_string = HtmlFragment('<b>test</b>')
```

```python
<test>
```

_class_rich_repr_(display_manager, **kwds)

Rich Output Magic Method

See sage.repl.rich_output for details.

EXAMPLES:

```
sage: from sage.misc.html import HtmlFragment
sage: h = HtmlFragment('<b>old</b>')
```

```python
# the doctest backend does not support html
```

```
OutputPlainText container
```

```
>>> from sage.all import *
>>> from sage.repl.rich_output import get_display_manager
>>> dm = get_display_manager()
```

```
OutputPlainText container
```

```
>>> from sage.misc.html import HtmlFragment
```

```
<test>
```

```
>>> from sage.misc.html import HtmlFragment
```

```
<test>
```

class sage.misc.html.MathJax

Bases: object

Render LaTeX input using MathJax. This returns a MathJaxExpr.

EXAMPLES:

```
sage: from sage.misc.html import MathJax
sage: MathJax()('3')
```

```
\[3\]
```

```
>>> from sage.misc.html import MathJax
```

```
\[3\]
```

```
>>> from sage.misc.html import MathJax
```

```
\[3\]
```

4.2. HTML and MathML
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 (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:

```python
sage: from sage.misc.html import MathJax
sage: MathJax().eval(3, mode='display')
<html>\[3\]</html>
sage: MathJax().eval(3, mode='inline')
<html>\(3\)</html>
sage: MathJax().eval(type(3), mode='inline')
<html>\(\verb|&lt;class|
\verb| |
\verb|sage.rings.integer.Integer>|\)</html>
```
Utilities, Release 10.4

```python
>>> from sage.all import *
>>> from sage.misc.html import MathJaxExpr
>>> MathJaxExpr("a^{2/3}") + MathJaxExpr("x^{-1}\")
a^{2/3}x^{-1}
```

sage.misc.html.html(obj, concatenate=True, strict=False)

Construct a HTML fragment

**INPUT:**

- obj – anything. An object for which you want an HTML representation.
- concatenate – if True, combine HTML representations of elements of the container obj
- strict – if True, construct an HTML representation of obj even if obj is a string

**OUTPUT:**

A HtmlFragment instance.

**EXAMPLES:**

```python
sage: h = html('<hr>'); pretty_print(h)
<hr>
sage: type(h)
<class 'sage.misc.html.HtmlFragment'>

sage: html(1/2)
<html>
\(\frac{1}{2}\)</html>
sage: html(<a href="http://sagemath.org">sagemath</a>)
sagemath
sage: html(<a href="http://sagemath.org">sagemath</a>, strict=True)
\verb|&lt;a| \verb|href="http://sagemath.org">sagemath&lt;/a|\)
```

Display preference align_latex affects rendering of LaTeX expressions:

```python
sage: from sage.repl.rich_output.display_manager import get_display_manager
sage: dm = get_display_manager()
sage: dm.preferences.align_latex = 'left'
sage: html(1/2)
```

(continues on next page)
Utilities, Release 10.4

\[
\displaystyle \frac{1}{2}
\]

sage:
```python
dm.preferences.align_latex = 'center'
```

sage:
```html
\frac{1}{2}
```

sage:
```python
dm.preferences.align_latex = None  # same with left
```

sage:
```html
\frac{1}{2}
```

```python
>>> from sage.all import *
>>> from sage.repl.rich_output.display_manager import get_display_manager

>>> dm = get_display_manager()

>>> dm.preferences.align_latex = 'left'

>>> html(Integer(1)/Integer(2))

\[
\frac{1}{2}
\]

>>> dm.preferences.align_latex = 'center'

>>> html(Integer(1)/Integer(2))

\[
\frac{1}{2}
\]

>>> dm.preferences.align_latex = None  # same with left

>>> html(Integer(1)/Integer(2))

\[
\frac{1}{2}
\]
```

sage.misc.html.math_parse(s)

Transform the string \( s \) with TeX maths to an HTML string renderable by MathJax.

INPUT:

- \( s \) – a string

OUTPUT:

A `HtmlFragment` instance.

Specifically this method does the following:

- Replace all \$text$'s by \\text{}
- Replace all $$text$$'s by \[text\]
- Replace all \$'s by $'s. Note that this has precedence over the above two cases.

EXAMPLES:

```python
sage: print(sage.misc.html.math_parse('This is $2+2$.'))
This is \(2+2\).

sage: print(sage.misc.html.math_parse('This is $$2+2$$.'))
This is \(2+2\).

sage: print(sage.misc.html.math_parse(r'$2+2$ is rendered to $2+2$.'))
<span>$2+2</span> is rendered to \(2+2\).
```

```python
>>> from sage.all import *

>>> print(sage.misc.html.math_parse('This is $2+2$.'))
This is \(2+2\).

>>> print(sage.misc.html.math_parse('This is $$2+2$$.'))
This is \(2+2\).

>>> print(sage.misc.html.math_parse('This is \[2+2\].'))
This is \[2+2\].
```

(continues on next page)
sage.misc.html.pretty_print_default(enable=True)
Enable or disable default pretty printing.

Pretty printing means rendering things in HTML and by MathJax so that a browser-based frontend can render real math.

This function is pretty useless without the notebook, it should not be in the global namespace.

INPUT:
• enable – bool (default: True). If True, turn on pretty printing; if False, turn it off.

EXAMPLES:

```python
sage: pretty_print_default(True)
sage: 'foo'  # the doctest backend does not support html
'foo'
sage: pretty_print_default(False)
sage: 'foo'
'foo'
```

```python
>>> from sage.all import *
>>> pretty_print_default(True)
>>> 'foo'  # the doctest backend does not support html
'foo'
>>> pretty_print_default(False)
>>> 'foo'
'foo'
```

### 4.2.2 MathML output support

In order to support MathML formatting, an object should define a special method _mathml_(self) that returns its MathML representation.

```python
class sage.misc.mathml.MathML
    Bases: str
    sage.misc.mathml.bool_function(x)
    sage.misc.mathml.list_function(x)
    sage.misc.mathml.mathml(x)
    Output x formatted for inclusion in a MathML document.
    sage.misc.mathml.str_function(x)
    sage.misc.mathml.tuple_function(x)
```
4.3 LaTeX

4.3.1 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 tutorial (this link should take you to a local copy of the tutorial).

4.3.2 LaTeX printing support

In order to support latex formatting, an object should define a special method \texttt{\_latex\_(self)} that returns a string, which will be typeset in a mathematical mode (the exact mode depends on circumstances).

This module focuses on using LaTeX for printing. For the use of LaTeX for rendering math in HTML by MathJax, see MathJax defined in \texttt{sage.misc.html}.

AUTHORS:

- William Stein: original implementation
- Joel B. Mohler: latex_variable_name() drastic rewrite and many doc-tests

class \texttt{sage.misc.latex.Latex(debug=False, slide=False, density=150, engine=None)}

Enter, e.g.,

```latex
\text{\texttt{latex}}
```
The equation $y^2 = x^3 + x$ defines an elliptic curve. We have $2006 = \text{sage(factor(2006))}$.

in an input cell in the notebook to get a typeset version. Use \texttt{%latex\_debug} to get debugging output.

Use \texttt{latex(...) to typeset a Sage object. Use LatexExpr to typeset LaTeX code that you create by hand. Use \texttt{%slide} instead to typeset slides.

\textbf{Warning:} You must have dvipng (or dvips and magick/convert) installed on your operating system, or this command will not work.

EXAMPLES:

```python
sage: latex(x^20 + 1)  #...
\text{\texttt{\_needs sage.symbolic}}
x^{(20)} + 1
sage: latex(FiniteField(25, 'a'))  #...
\text{\texttt{\_needs sage.rings.finite_rings}}
\textbackslash{Bold}(F)_{\{5^{\{2\}}}\texttt{}}
sage: latex("hello")
\texttt{\textbackslash{texttt{\{hello\}}}}
sage: LatexExpr(r"\frac{x^2 - 1}{x + 1} = x - 1")
\frac{x^2 - 1}{x + 1} = x - 1
```

458 Chapter 4. Formatted Output
LaTeX expressions can be added; note that a space is automatically inserted:

```
from sage.all import *
latex(x**Integer(20) + Integer(1))    # needs sage.symbolic
x^{20} + 1
latex(FiniteField(Integer(25), 'a'))  # needs sage.rings.finite_rings
\Bold{F}_{5^{2}}
latex("hello")
\text\{\texttt{hello}\}
LatexExpr(r"\frac{x^2 - 1}{x + 1} = x - 1")
\frac{x^2 - 1}{x + 1} = x - 1
```

**add_macro** *(macro)*

Append to the string of extra LaTeX macros, for use with %latex and %html.

**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()**
add_to_preamble(s)

Append to the string s of extra LaTeX macros, for use with $\LaTeX$.

EXAMPLES:

```
sage: latex.extra_preamble()
```
```
sage: latex.add_to_preamble("\\DeclareMathOperator{Ext}{Ext}\")
```

```from sage.all import *
>>> latex.extra_preamble()
>>> latex.add_to_preamble("\\DeclareMathOperator{Ext}{Ext}\")
```

At this point, a notebook cell containing

```
\[
\Ext_A^* (\GF{2}, \GF{2}) \Rightarrow \pi_*^* (S^0)
\]
```

will be typeset correctly.

```
sage: latex.add_to_preamble("\\usepackage{xypic}"
```
```
sage: latex.extra_preamble()
```
```
\"\\\DeclareMathOperator{Ext}{Ext}\\usepackage{xypic}\"
```

```
\[
\begin{array}{cccc}
\circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ \\
\end{array}
\]
```

Now one can put various xypic diagrams into a $\LaTeX$ cell, such as

```
\[
\begin{array}{cccc}
\circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ \\
\end{array}
\]
```

Reset the preamble to its default, the empty string:

```
sage: latex.extra_preamble('')
```
```
sage: latex.extra_preamble()
```
```
\[
\begin{array}{cccc}
\circ & \circ & \circ & \circ \\
\circ & \circ & \circ & \circ \\
\end{array}
\]
```

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:

```python
sage: latex.blackboard_bold()
False
sage: latex.blackboard_bold(True)
sage: latex.blackboard_bold()
True
sage: latex.blackboard_bold(False)
```

```python
>>> from sage.all import *
>>> latex.check_file("article.cls")
# optional - latex
```

```python
sage: latex.check_file("some_inexistent_file.sty")
Warning: `some_inexistent_file.sty` is not part of this computer's TeX... installation.
```

```python
sage: latex.check_file("some_inexistent_file.sty")
sage: latex.check_file("some_inexistent_file.sty", "This file is required for... installation.
```

```python
This file is required for blah. It can be downloaded from: http://blah.org/
```

This test checks that the bug in Issue #9091 is fixed:
sage: latex.check_file("article.cls", "The article class is really critical.") # optional - latex

```python
>>> from sage.all import *
```  

```python
>>> latex.check_file("article.cls", "The article class is really critical.") # optional - latex
```

**engine** *(e=None)*

Set Sage to use `e` as latex engine when typesetting with `view()`, in `%latex` cells, etc.

**INPUT:**

- `e` = 'latex', 'pdflatex', 'xelatex', 'lualatex' 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:

```python
latex.extra_preamble(r'''
\usepackage{fontspec,xunicode,xltxtra}
\setmainfont{some font here}
\setmonofont{another font here}''')
```

**EXAMPLES:**

```python
sage: latex.engine()  # random
'lualatex'
```

```python
sage: latex.engine("latex")
```

```python
sage: latex.engine()  # random
'latex'
```

```python
sage: latex.engine("pdflatex")
```

```python
sage: latex.engine()  # random
'pdflatex'
```

```python
>>> from sage.all import *
```  

```python
>>> latex.engine()  # random
'lualatex'
```  

```python
>>> latex.engine("latex")
```  

```python
>>> latex.engine()  # random
'latex'
```  

```python
>>> latex.engine("pdflatex")
```  

```python
>>> latex.engine()  # random
'pdflatex'
```

**eval** *(x, globals=None, strip=False, filename=None, debug=None, density=None, engine=None, locals={})*

Compile 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
• engine – latex engine to use. Currently 'latex', 'pdflatex', 'xelatex' and 'lualatex' 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 magick/convert) installed on your operating system, or this command will not work. When using 'pdflatex', 'xelatex' or 'lualatex', you must have magick/convert installed.

**OUTPUT:**

If it compiled successfully, this returns an empty string '', otherwise it returns None.

**EXAMPLES:**

```python
sage: fn = tmp_filename()
sage: latex.eval("$\ZZ[x]$", locals(), filename=fn) # not tested
''
```

```python
sage: latex.eval(r"\ThisIsAnInvalidCommand", {}) # optional -- latex...

An error occurred...
No pages of output...
```

```python
>>> from sage.all import *
>>> latex.eval("$\ZZ[x]$", locals(), filename=fn) # not tested
''
```

```python
>>> latex.eval(r"\ThisIsAnInvalidCommand", {}) # optional -- latex ImageMagick
An error occurred...
No pages of output...
```

**extra_macros (macros=None)**

String containing extra LaTeX macros to use with %latex and %html.

**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:**

```python
sage: latex.extra_macros("""
newcommand{\foo}{bar}"
"
"
```

```python
sage: latex.extra_macros() 'newcommand{\foo}{bar}'
sage: latex.extra_macros(""")
sage: latex.extra_macros()
''
```

```python
>>> from sage.all import *
>>> latex.extra_macros("""
newcommand{\foo}{bar}"
"
"
```

```python
>>> latex.extra_macros() 'newcommand{\foo}{bar}'
>>> latex.extra_macros(""")
>>> latex.extra_macros()
''
```
**extra_preamble** *(s=None)*

String containing extra preamble to be used with `%latex`.

**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:**

```python
sage: latex.extra_preamble(r'\\DeclareMathOperator{\Ext}{Ext}')
sage: latex.extra_preamble('\\DeclareMathOperator{\Ext}{Ext}')
sage: latex.has_file('article.cls')
True
sage: latex.has_file('some_inexistent_file.sty')
False
```
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:**

```python
sage: # needs sage.modules
sage: a = matrix([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)

sage: latex.matrix_column_alignment('r')
```

```python
>>> from sage.all import *

>>> a = matrix(Integer(1), Integer(1), [Integer(42)])

>>> latex(a)
\left(\begin{array}{r}
42
\end{array}\right)

>>> latex.matrix_column_alignment('c')

>>> latex(a)
\left(\begin{array}{c}
42
\end{array}\right)

>>> latex.matrix_column_alignment('l')

>>> latex(a)
\left(\begin{array}{l}
42
\end{array}\right)

>>> latex.matrix_column_alignment('r')
```

Restore defaults:

```python
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: # needs sage.modules
sage: a = matrix(1, 1, [17])
sage: latex(a)
\left(\begin{array}{r}
17 \\
\end{array}\right)
sage: latex.matrix_delimiters('[', ']

(continues on next page)
Utilities, Release 10.4

>>> latex.matrix_delimiters('[', ']')
>>> latex(a)
\left[\begin{array}{r}17
\end{array}\right]
>>> latex.matrix_delimiters(left='\{')
>>> latex(a)
\left\{\begin{array}{r}17
\end{array}\right\}
>>> latex.matrix_delimiters()
[\{, \}]

Restore defaults:

sage: latex.matrix_delimiters('(', ')')

sage: from sage.all import *
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: # needs sage.modules
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]

(continues on next page)
```python
sage: latex(a)
\left\{1, 2, 3\right\}
sage: latex.vector_delimiters()
[', ', ']

>>> from sage.all import *

>>> # needs sage.modules

>>> a = vector(QQ, [Integer(1), Integer(2), Integer(3)])

>>> latex(a)
\left(1, 2, 3\right)

>>> latex.vector_delimiters(['', ''])

>>> latex(a)
\left[1, 2, 3\right]

>>> latex.vector_delimiters(right=\)

>>> latex(a)
\left[1, 2, 3\right]

>>> latex.vector_delimiters()
[', ', ']

```

```text
Restore defaults:

```
```python
sage: latex.vector_delimiters(('', ''))

```text
```python
>>> from sage.all import *

>>> latex.vector_delimiters(('', ''))

```

### class sage.misc.latex.LatexCall

**Bases:** object

Typeset Sage objects via a `__call__` method to this class, typically by calling those objects' `_latex_` methods. The class `LatexCall` 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:**

```python
sage: from sage.misc.latex import LatexCall

sage: LatexCall()(ZZ)
\Bold{Z}

sage: LatexCall().__call__(ZZ)
\Bold{Z}

```

```text
This returns an instance of the class `LatexExpr`:

```
```python
sage: type(LatexCall()(ZZ))
<class 'sage.misc.latex.LatexExpr'>

```text
```python
>>> from sage.all import *

>>> type(LatexCall()(ZZ))
<class 'sage.misc.latex.LatexExpr'>

```

468 Chapter 4. Formatted Output
class sage.misc.latex.LatexExamples
Bases: object

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:

```python
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)}}}}
```

```python
>>> from sage.all import *
>>> from sage.misc.latex import latex_examples
>>> K = latex_examples.knot()
>>> K
LaTeX example for testing display of a knot produced by xypic...
>>> 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: sageobject

LaTeX example for testing display of commutative diagrams. See its string representation for details.

EXAMPLES:

```python
sage: from sage.misc.latex import latex_examples
sage: CD = latex_examples.diagram()
sage: CD
LaTeX example for testing display of a commutative diagram...
```

```python
>>> from sage.all import *
>>> from sage.misc.latex import latex_examples
>>> CD = latex_examples.diagram()
>>> CD
LaTeX example for testing display of a commutative diagram...
```

class graph
Bases: 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...

>>> from sage.all import *
>>> from sage.misc.latex import latex_examples
>>> G = latex_examples.graph()
>>> G
LaTeX example for testing display of graphs...

class knot

Bases: 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...

>>> from sage.all import *
>>> from sage.misc.latex import latex_examples
>>> K = latex_examples.knot()
>>> K
LaTeX example for testing display of a knot...

class pstricks

Bases: 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...

>>> from sage.all import *
>>> from sage.misc.latex import latex_examples
>>> PS = latex_examples.pstricks()
>>> 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 Jupyter notebook, use pretty_print() 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:**

```python
sage: latex(x^20 + 1)  # needs sage.symbolic
x^{20} + 1
sage: LatexExpr(r"\frac{x^2 + 1}{x - 2}")  # needs sage.symbolic
\frac{x^2 + 1}{x - 2}
```

`LatexExpr` simply converts to string without doing anything extra, it does *not* call `latex()`:

```python
sage: latex(ZZ)  \Bold{Z}
sage: LatexExpr(ZZ)  \text{Integer Ring}
```

The result of `latex()` is of type `LatexExpr`:

```python
sage: L = latex(x^20 + 1)  # needs sage.symbolic
```

A `LatexExpr` can be converted to a plain string:
Utilities, Release 10.4

```
sage: str(latex(x^20 + 1)) # needs sage.symbolic
'x^{20} + 1'
```

```
>>> from sage.all import *
>>> str(latex(x**Integer(20) + Integer(1))) # needs sage.symbolic
'x^{20} + 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}
```

```
>>> from sage.all import *
>>> from sage.misc.latex import None_function
>>> 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}
```
```
>>> from sage.all import *
>>> from sage.misc.latex import bool_function
>>> print(bool_function(Integer(2)==Integer(3)))
\mathrm{False}
>>> print(bool_function(Integer(3)==(Integer(2)+Integer(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:

```python
sage: from sage.misc.latex import builtin_constant_function
sage: builtin_constant_function(True)
'\text{\texttt{True}}'
sage: builtin_constant_function(None)
'\text{\texttt{None}}'
sage: builtin_constant_function(NotImplemented)
'\text{\texttt{NotImplemented}}'
sage: builtin_constant_function(Ellipsis)
'\text{\texttt{Ellipsis}}'
```

```python
>>> from sage.all import *
>>> from sage.misc.latex import builtin_constant_function
>>> builtin_constant_function(True)
'\text{\texttt{True}}'
>>> builtin_constant_function(None)
'\text{\texttt{None}}'
>>> builtin_constant_function(NotImplemented)
'\text{\texttt{NotImplemented}}'
>>> builtin_constant_function(Ellipsis)
'\text{\texttt{Ellipsis}}'
```

`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:**

```python
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)  # needs sage.symbolic
'\left(-x^{2}\right)'
```

```python
>>> from sage.all import *
>>> from sage.misc.latex import coeff_repr
>>> coeff_repr(QQ(Integer(1)/Integer(2)))
'\frac{1}{2}'
>>> coeff_repr(-x**Integer(2))  # needs sage.symbolic
'\left(-x^{2}\right)'
```

`sage.misc.latex.default_engine()`

Return the default latex engine and the official name of the engine.

This is determined by availability of the popular engines on the user's system. It is assumed that at least latex is available.

**EXAMPLES:**
sage: from sage.misc.latex import default_engine
default_engine()  # random ('lualatex', 'LuaLaTeX')

>>> from sage.all import *
>>> from sage.misc.latex import default_engine
>>> default_engine()  # random ('lualatex', 'LuaLaTeX')

describe

sage: x,y,z = var('x,y,z')

sage.misc.latex.dict_function(x)

Return the LaTeX code for a dictionary $x$.

INPUT:

• $x$ – a dictionary

EXAMPLES:

sage: # needs sage.symbolic
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]}

latex(d)
\left\{\left(1, 2, x^{2}\right) : \left[\sin\left(z^{2}\right), \frac{1}{2} \, y\right]\right\}

sage: x,y,z = var('x,y,z')

sage.misc.latex.float_function(x)

Returns the LaTeX code for a python float $x$.

INPUT:

• $x$ – a python float

EXAMPLES:

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
```python
>>> from sage.all import *
>>> from sage.misc.latex import float_function
>>> float_function(float(RealNumber('3.14')))
3.14
>>> float_function(float(RealNumber('1e-10')))
1 \times 10^{-10}
>>> float_function(float(RealNumber('2e10')))
20000000000.0
```

`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:

```python
sage: from sage.misc.latex import has_latex_attr
sage: has_latex_attr(identity_matrix(3)) # needs sage.modules
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:

```python
sage: # needs sage.modules
sage: T = type(identity_matrix(3)); T
<class 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
sage: hasattr(T, '_latex_')
True
sage: T._latex_()
Traceback (most recent call last):
  ..._latex_... needs an argument
sage: has_latex_attr(T)
False
```

```python
sage: # needs sage.modules
sage: T = type(identity_matrix(Integer(3))); T
<class 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
sage: hasattr(T, '_latex_')
True
sage: T._latex_()
Traceback (most recent call last):
  ..._latex_... needs an argument
sage: has_latex_attr(T)
False
```

`sage.misc.latex.latex(x, combine_all=False)`
Return a \texttt{LatexExpr} built out of the argument \( x \).

\textbf{INPUT:}

- \( x \) – a Sage object
- \texttt{combine\_all} – boolean (Default: False) If \texttt{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.

\textbf{OUTPUT:}

A \texttt{LatexExpr} built from \( x \)

\textbf{EXAMPLES:}

\begin{verbatim}
sage: latex(Integer(3))  # indirect doctest
3
sage: latex(1==0)
\texttt{\textbackslash mathrm\{False\}}
sage: print(latex([x, 2]))
\texttt{\textbackslash left\{x, 2\textbackslash right\}}

>>> from sage.all import *

>>> latex((x,Integer(2)), combine_all=True)
\texttt{x 2}

sage.misc.latex.latex_extra_preamble()

Return the string containing the user-configured preamble, \texttt{sage\_latex\_macros}, and any user-configured macros. This is used in the \texttt{eval()} method for the \texttt{Latex} class, and in \texttt{\_latex\_file\_}(); it follows either \texttt{LATEX\_HEADER} or \texttt{SLIDE\_HEADER} (defined at the top of this file) which is a string containing the document-class and standard usepackage commands.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.misc.latex import latex_extra_preamble

sage: print(latex_extra_preamble())
\newcommand\{\ZZ\}\{\texttt{\textbackslash Bold}(Z)\}
\newcommand\{\NN\}\{\texttt{\textbackslash Bold}(N)\}
\newcommand\{\RR\}\{\texttt{\textbackslash Bold}(R)\}
\newcommand\{\CC\}\{\texttt{\textbackslash Bold}(C)\}
(continues on next page)
\end{verbatim}
\newcommand{\QQ}{\Bold{Q}}
\newcommand{\QQbar}{\overline{\QQ}}
\newcommand{\GF}{[1]}{\Bold{F}_{#1}}
\newcommand{\Zp}{[1]}{\Bold{Z}_{#1}}
\newcommand{\Qp}{[1]}{\Bold{Q}_{#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{\SL}{\mathrm{SL}}
\newcommand{\PSL}{\mathrm{PSL}}
\newcommand{\lcm}{\mathop{\operatorname{lcm}}}
\newcommand{\dist}{\mathrm{dist}}
\newcommand{\Bold}{[1]}{\mathbf{#1}}

>>> from sage.all import *
>>> from sage.misc.latex import latex_variable_name

>>> print(latex_variable_name(x, is_fname=False))

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:**

```python
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}
```

```python
>>> from sage.all import *
>>> from sage.misc.latex import latex_variable_name
>>> latex_variable_name(a)
a
>>> latex_variable_name(abc)
\mathit{abc}
>>> latex_variable_name(sigma)
\sigma
>>> latex_variable_name(sigma_k)
\sigma_{k}
>>> latex_variable_name(sigma389)
\sigma_{389}
>>> latex_variable_name(beta_00)
\beta_{(00)}
>>> latex_variable_name(Omega84)
\Omega_{84}
>>> latex_variable_name(sigma_alpha)
\sigma_{\alpha}
>>> latex_variable_name(nothing1)
\mathit{nothing}_{1}
>>> latex_variable_name(nothing1, is_fname=True)
{\rm nothing}_{1}
>>> latex_variable_name(nothing_abc)
\mathit{nothing}_{\mathit{abc}}
>>> latex_variable_name(nothing_abc, is_fname=True)
{\rm nothing}_{{\rm abc}}
>>> latex_variable_name(alpha_beta_gamma12)
\alpha_{\beta_{\gamma_{12}}}
>>> latex_variable_name(x_ast)
x_{\ast}
```

(continues on next page)
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 “\unbox a” if “is_fname” flag is True or False.

INPUT:
- a – string

OUTPUT: A string

EXAMPLES:

```python
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'
```

sage.misc.latex.list_function(x)

Returns the LaTeX code for a list x.

INPUT:
- x – a list

EXAMPLES:

```python
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
```

(continues on next page)
\left[1, 2, 3\right]

sage: latex([Matrix(ZZ, 3, range(9)), # indirect doctest
         \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)

>>> from sage.all import *
>>> from sage.misc.latex import list_function

>>> list_function([Integer(1),Integer(2),Integer(3)])
\left[1, 2, 3\right]

>>> latex([Integer(1),Integer(2),Integer(3)])
\left[1, 2, 3\right]

>>> latex([Matrix(ZZ, Integer(3), range(Integer(9))), # indirect doctest
         \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, engine=None)

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
- engine – (default: None) 'latex', 'pdflatex', 'xelatex' or 'lualatex'

EXAMPLES:
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:**

```python
sage: t = PolynomialRing(QQ, 't').gen(0)
sage: from sage.misc.latex import repr_lincomb
sage: repr_lincomb(['a', 's', ''], [-t, t - 2, t^12 + 2])
'\text{\texttt{a}} + \left(t - 2\right)\text{\texttt{s}} + \left(t^{12} + 2\right)'
```

Verify that a certain corner case works (see Issue #5707 and Issue #5766):

```python
sage: repr_lincomb([1,5,-3],[2,8/9,7])
'2\cdot 1 + \frac{8}{9}\cdot 5 + 7\cdot -3'
```

Verify that Issue #17299 (latex representation of modular symbols) is fixed:

```python
sage: x = EllipticCurve('64a1').modular_symbol_space(sign=1).basis()[0]
from sage.misc.latex import repr_lincomb
sage: latex(x.modular_symbol_rep())
```

(continued on next page)
\NeedsSageSchemes
\left\langle \frac{-3}{11}, \frac{-1}{4} \right\rangle - \left\langle \frac{3}{13}, \frac{1}{4} \right\rangle

\begin{verbatim}
from sage.all import *

x = EllipticCurve('64a1').modular_symbol_space(sign=Integer(1)).basis()[Integer(0)]
# needs sage.schemes
latex(x.modular_symbol_rep())
# needs sage.schemes
\end{verbatim}

Verify that it works when the symbols are numbers:

\begin{verbatim}
sage: x = FormalSum([(1,2),(3,4)])
sage: latex(x)
2 + 3\cdot 4
\end{verbatim}

\begin{verbatim}
from sage.all import *

x = FormalSum([(Integer(1),Integer(2)),(Integer(3),Integer(4))])
latex(x)
2 + 3\cdot 4
\end{verbatim}

Verify that it works when \texttt{bv} in \texttt{CC} raises an error:

\begin{verbatim}
sage: x = FormalSum([(1,\texttt{x}), (2,\texttt{y})])
sage: latex(x)
\texttt{x} + 2\texttt{y}
\end{verbatim}

\begin{verbatim}
from sage.all import *

x = FormalSum([(Integer(1),\texttt{x}), (Integer(2),\texttt{y})])
latex(x)
\texttt{x} + 2\texttt{y}
\end{verbatim}

\begin{verbatim}
sage.misc.latex.str_function(x)
\end{verbatim}

Return a LaTeX representation of the string \texttt{x}.

The main purpose of this function is to generate LaTeX representation for classes that do not provide a customized method.

If \texttt{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 \texttt{x} is wrapped in a \texttt{\texttt{ttt}} command and these lines are assembled in a left-justified array. This gives to complicated strings the closest look to their “terminal representation”.

\textbf{Warning}: Such wrappers \texttt{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.

\begin{itemize}
  \item \texttt{x} – a string
\end{itemize}

\textbf{OUTPUT}: A string
EXAMPLES:

```python
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\_world}}'
sage: str_function('-1.00000?')  # trac 12178
'-1.00000?'
```

```python
>>> from sage.all import *
>>> from sage.misc.latex import str_function
>>>
```

sage.misc.latex.\texttt{tuple_function}(x, combine\_all=False)

Returns the LaTeX code for a tuple \( x \).

INPUT:

- \( x \) – a tuple
- \texttt{combine\_all} – boolean (default: False) If \texttt{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:

```python
sage: from sage.misc.latex import tuple_function
sage: tuple_function((1,2,3))
'\left(1, 2, 3\right)'
```

```python
>>> from sage.all import *
>>> from sage.misc.latex import tuple_function
>>>
```

Check that Issue \#11775 is fixed:

```python
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'
```
Utilities, Release 10.4

```python
>>> from sage.all import *

>>> tuple_function((Integer(1),Integer(2),Integer(3)), combine_all=True)
'1 2 3'

>>> tuple_function(((Integer(1),Integer(2)),Integer(3)), combine_all=True)
'\left(1, 2\right) 3'
```

`sage.misc.latex.view(objects, title='Sage', debug=False, sep='', tiny=False, engine=None, viewer=None, tightpage=True, margin=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.
- `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`
  - 'lualatex' - 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 hboxes are detected.
- `viewer` - string or None (default: None); specify a viewer to use; currently the only options are None and 'pdf'.
- `tightpage` - bool (default: True); use the LaTeX package `preview` with the 'tightpage' option.
- `margin` - float or None (default: None); adds a margin of margin mm; has no effect if the option `tightpage` is False.
- `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.

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 the engine is either 'pdflatex', 'xelatex', or 'lualatex', 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 (this is the default setting) 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

```latex
\usepackage[tightpage,active]{preview}
\PreviewEnvironment{page}
```

to the LaTeX preamble, and replaces the \{ and \} around each object by \begin{page}$ and $\end{page}. Setting `tightpage` to False turns off this behavior and provides the latex output as a full page. If `tightpage` is set to True, the Title is ignored.

### 4.3.3 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_docbuild.conf` 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.

Any macro defined here may be used in doctstrings or in the tutorial (or other pieces of documentation). In a doctstring, 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/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


EXAMPLES:

```python
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}{\Bold{F}_{#1}}) ('GF', ['\Bold{F}_{#1}', 1])
```

```python
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:

```python
sage: from sage.misc.latex_macros import produce_latex_macro
sage: produce_latex_macro('ZZ') '\newcommand{\ZZ}{\Bold{Z}}'
```

```python
sage: produce_latex_macro('GF', 37) '\newcommand{\GF}{\Bold{F}_{#1}}'
```

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:

```python
sage: produce_latex_macro('GF', Integer(37)) '\newcommand{\GF}{\Bold{F}_{#1}}'
```

If the Sage object is not in the global name space, describe it like so:

```python
sage: produce_latex_macro('sage.rings.finite_rings.finite_field_constructor.
FiniteField', 3) '\newcommand{\FiniteField}{\Bold{F}_{#1}}'
```
4.3.4 Standalone LaTeX Document class and TikzPicture

This module contains two Python classes. Firstly, it contains a class `Standalone` to represent a LaTeX file using the `standalone` document class.

From its documentation:

> The standalone bundle allows users to easily place picture environments or other material in own source files and compile these on their own or as part of a main document. A special standalone class is provided for use with such files, which by default crops the resulting output file to the content. The standalone package enables the user to simply load the standalone files using “input” inside a main document.

Secondly, it contains a class `TikzPicture` which inherits from `Standalone` that represents a LaTeX file using the standalone document class and containing a tikzpicture.

A Python Module for PGF/Tikz pictures. A TikzPicture object is created from a string starting with `r'\begin{tikzpicture}'` and ending with `r'\end{tikzpicture}'`.

The module allows to convert a standalone LaTeX document class file, including tikzpictures, to an image. It allows conversion to pdf, png and svg formats. It also show them automatically in Jupyter using rich representation.
According to wikipedia, PGF/TikZ is a pair of languages for producing vector graphics (e.g., technical illustrations and drawings) from a geometric/algebraic description, with standard features including the drawing of points, lines, arrows, paths, circles, ellipses and polygons.

EXAMPLES:

**Standalone LaTeX document class**

First *Hello World* example:

```latex
\documentclass{standalone}
\begin{document}
Hello World
\end{document}
```

Loading a few latex packages:

```latex
\documentclass{standalone}
\usepackage{amsmath}
\usepackage{amsfont}
\begin{document}
Hello World
\end{document}
```

Setting few standalone options (see documentation of standalone for a complete list):

```latex
\documentclass{standalone}
\standaloneconfig{border=4mm}
\standaloneconfig{beamer=true}
\begin{document}
Hello World
\end{document}
```
Adding your own list of macros:

```latex
\documentclass{standalone}
\newcommand{\ZZ}{\mathbb{Z}}
\begin{document}
Hello World
\end{document}
```

It provides conversion to images of different format:

```python
>>> from sage.all import *
>>> from sage.misc.latex_standalone import Standalone
>>> s = Standalone('Hello World')
>>> _ = s.pdf()  # not tested
>>> _ = s.png()  # not tested
>>> _ = s.svg()  # not tested
>>> s  # not tested, in Jupyter, this shows the image directly below the cell
```

**TikzPicture**

This module also contains a class `TikzPicture` which inherits from `Standalone` to represent more specifically a tikzpicture which is within a standalone document class.

First construct a string describing a tikzpicture:

```python
sage: lines = []
sage: lines.append(r'\begin{tikzpicture}"
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

```sage
s = \n
.join(lines)
sage: print(s)
\begin{tikzpicture}
\draw[very thick,orange,->] (0,0) -- (1,1);
\end{tikzpicture}
```

```python
globals() = load_sage() =
lines = []
lines.append(r\begin{tikzpicture})
lines.append(r\draw[very thick,orange,->] (0,0) -- (1,1);)
lines.append(r\end{tikzpicture})
s = \n
.join(lines)
sage: print(s)
\begin{tikzpicture}
\draw[very thick,orange,->] (0,0) -- (1,1);
\end{tikzpicture}
```

One may provide it as input to `TikzPicture`:

```sage
sage: from sage.misc.latex_standalone import TikzPicture
sage: t = TikzPicture(s)
```

In the terminal, the following shows the content of the standalone document class `tex` file which contains the `tikzpicture`. In Jupyter, it shows the picture itself:

```sage
t
\documentclass[tikz]{standalone}
\begin{document}
\begin{tikzpicture}
\draw[very thick,orange,->] (0,0) -- (1,1);
\end{tikzpicture}
\end{document}
```

As it is the case for `Standalone`, the constructor of `TikzPicture` has many arguments allowing for example to add some more `\usepackage` lines:

```sage
t = TikzPicture(s, usepackage=['amsmath'])
sage: t
\documentclass[tikz]{standalone}
\usepackage{amsmath}
\begin{document}
\begin{tikzpicture}
\draw[very thick,orange,->] (0,0) -- (1,1);
\end{tikzpicture}
\end{document}
```

(continues on next page)
usepackage{amsmath}
\begin{document}
\begin{tikzpicture}
\draw[very thick,orange,->] (0,0) -- (1,1);
\end{tikzpicture}
\end{document}

Moreover, it allows to load some tikz libraries:

\begin{verbatim}
\documentclass[tikz]{standalone}
\usetikzlibrary{arrows}
\begin{document}
\begin{tikzpicture}
\draw[very thick,orange,->] (0,0) -- (1,1);
\end{tikzpicture}
\end{document}
\end{verbatim}

The following example illustrates that it works when providing the \texttt{tikzpicture} code generated by Sage from some polyhedron:

\begin{verbatim}
\documentclass[tikz]{standalone}
\begin{document}
\begin{tikzpicture}
\draw[very thick,orange,->] (0,0) -- (1,1);
\end{tikzpicture}
\end{document}
\end{verbatim}

(continues on next page)
... \[Integer(0), Integer(1), Integer(0)]\], \[-\text{Integer}(1), Integer(0), Integer(0)]\],
\[-\text{Integer}(0), Integer(1), Integer(1)]\], \[Integer(0), Integer(0), Integer(1)]\], \[-\text{Integer}(0),\]
\[-\text{Integer}(1), Integer(0)]\]\]

>>> \text{P} = \text{Polyhedron}((\text{vertices}=\text{V}).\text{polar})

# needs sage.geometry.polyhedron

>>> \text{s1} = \text{P}.\text{projection()}.\text{tikz}(\text{[Integer(674), Integer(108), \text{-Integer}(731)]}, \text{Integer(112)},
\text{output_type}='\text{LaTeXExpr}') # needs sage.geometry.polyhedron sage.plot

>>> \text{t1} = \text{TikzPicture}(\text{s1})

# needs sage.geometry.polyhedron sage.plot

Open the image in a viewer (the returned value is a string giving the absolute path to the file in some temporary directory):

```python
sage: \text{path_to_file} = \text{t1}.\text{pdf()}
# not tested

# needs sage.geometry.polyhedron sage.plot
```

Instead, you may save a pdf of the tikzpicture into a file of your choice (but this does not open the viewer):

```python
sage: \_ = \text{t1}.\text{pdf}(\text{tikz_polytope.pdf})
# not tested

# needs sage.geometry.polyhedron sage.plot
```

Opening the image in a viewer can be turned off:

```python
sage: \_ = \text{t1}.\text{pdf}(\text{view}=\text{False})
# long time (2s), optional - latex, needs...

# sage.geometry.polyhedron sage.plot
```

The same can be done with png format (translated from pdf with convert command which needs the installation of imagemagick):

```python
sage: \_ = \text{t1}.\text{png}(\text{view}=\text{False})
# long time (2s), optional - imagemagick...

# latex, needs sage.geometry.polyhedron sage.plot
```

The string representation gives the header (5 lines) and tail (5 lines) of the tikzpicture. In Jupyter, it will instead use rich representation and show the image directly below the cell in png or svg format:

```python
sage: \text{t1}

# needs sage.geometry.polyhedron sage.plot
\documentclass[tikz]{standalone}
\begin{document}
\begin{tikzpicture}
(continues on next page)
```
Utilities, Release 10.4

>>> from sage.all import *

The current latex representation of a transducer is a tikzpicture using the tikz library automata. The string can be used as input:

Adding a border in the options avoids cropping the vertices of a graph:

The complete content of the file can be seen using `print(t)`:
AUTHORS:

• Sébastien Labbé, initial version in slabbe-0.2.spkg, nov 2015.
• Sébastien Labbé, inclusion into SageMath from slabbe-0.6.2, July 2021.

class sage.misc.latex_standalone.Standalone(content, document_class_options=None, standalone_config=None, usepackage=None, macros=None, use_sage_preamble=False)

Bases: SageObject

LaTeX standalone document class.

INPUT:

• content – string, the content to be added in the document between lines r'\begin{document}' and r'\end{document}'
• document_class_options – list of strings (default: []), latex document class standalone options. Such options appear on the line \documentclass[...]{standalone} between the brackets.
• standalone_config – list of strings (default: []), standalone configuration options. Such options are defined with \standaloneconfig{...}
• usepackage – list of strings (default: []), latex packages.
• macros – list of strings (default: []), stuff you need for the picture.
• use_sage_preamble – bool (default: False), whether to include sage latex preamble and sage latex macros, that is, the content of sage.misc.latex.extra_preamble(), sage.misc.latex.extra_macros() and sage.misc.latex_macros.sage_latex_macros().

EXAMPLES:
\documentclass{standalone}
\begin{document}
\section{Intro}
Test
\end{document}

```
sage: t = Standalone(content, standalone_config=['\texttt{border=4mm}'],
.....: usepackage=['amsmath'])
sage: t
\documentclass{standalone}
\standaloneconfig{border=4mm}
\usepackage{amsmath}
\begin{document}
\section{Intro}
Test
\end{document}
```

```python
>>> from sage.all import *

>>> t = Standalone(content, standalone_config=['\texttt{border=4mm}'],
.....: usepackage=['amsmath'])

>>> t
\documentclass{standalone}
\standaloneconfig{border=4mm}
\usepackage{amsmath}
\begin{document}
\section{Intro}
Test
\end{document}
```

**add_document_class_option**(option)

Add a document class option

**INPUT:**

- option – string

**EXAMPLES:**

```
sage: from sage.misc.latex_standalone import Standalone
sage: t = Standalone('Hello World')
sage: t.add_document_class_option('beamer')
sage: t
\documentclass[beamer]{standalone}
\begin{document}
Hello World
\end{document}
```

```python
>>> from sage.all import *

>>> from sage.misc.latex_standalone import Standalone

>>> t = Standalone('Hello World')

>>> t.add_document_class_option('beamer')

>>> t
\documentclass[beamer]{standalone}
\begin{document}
Hello World
\end{document}
```
**add_macro** (*macro*)

Add a macro

INPUT:

- `macro` – string, newcommand line

EXAMPLES:

```sage
from sage.misc.latex_standalone import Standalone
t = Standalone('Hello World')
t.add_macro(r'\newcommand{\ZZ}{\mathbb{Z}}')
t
\documentclass{standalone}
\newcommand{\ZZ}{\mathbb{Z}}
\begin{document}
Hello World
\end{document}
```

```python
>>> from sage.all import *
>>> from sage.misc.latex_standalone import Standalone
>>> t = Standalone('Hello World')
>>> t.add_macro(r'\newcommand{\ZZ}{\mathbb{Z}}')
>>> t
\documentclass{standalone}
\newcommand{\ZZ}{\mathbb{Z}}
\begin{document}
Hello World
\end{document}
```

**add_standalone_config** (*config*)

Add a standalone config

INPUT:

- `config` – string

EXAMPLES:

```sage
from sage.misc.latex_standalone import Standalone
t = Standalone('Hello World')
t.add_standalone_config("border=4mm")
t
\documentclass{standalone}
\standaloneconfig{border=4mm}
\begin{document}
Hello World
\end{document}
```

```python
>>> from sage.all import *
>>> from sage.misc.latex_standalone import Standalone
>>> t = Standalone('Hello World')
>>> t.add_standalone_config("border=4mm")
>>> t
\documentclass{standalone}
\standaloneconfig{border=4mm}
\begin{document}
Hello World
\end{document}
```
add_usepackage\(\text{\texttt{\textsc{package}}}\)
Add a usepackage line

INPUT:

• package – string, name of package

EXAMPLES:

\begin{verbatim}
sage: from sage.misc.latex_standalone import Standalone
t = Standalone('Hello World')
t.add_usepackage('amsmath')
t
\documentclass{standalone}
\usepackage{amsmath}
\begin{document}
Hello World
\end{document}
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> from sage.misc.latex_standalone import Standalone
>>> t = Standalone('Hello World')
>>> t.add_usepackage('amsmath')
>>> t
\documentclass{standalone}
\usepackage{amsmath}
\begin{document}
Hello World
\end{document}
\end{verbatim}

content() 
Return the content of the standalone document class file

EXAMPLES:

\begin{verbatim}
sage: from sage.misc.latex_standalone import Standalone
t = Standalone('Hello World')
t.content()
'Hello World'
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> from sage.misc.latex_standalone import Standalone
>>> t = Standalone('Hello World')
>>> t.content()
'Hello World'
\end{verbatim}

\begin{verbatim}
sage: from sage.misc.latex_standalone import TikzPicture
t = TikzPicture(s)
print(t.content())
\begin{tikzpicture}
draw (0,0) -- (1,1);
\end{tikzpicture}
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> from sage.misc.latex_standalone import TikzPicture
>>> s = "\begin{tikzpicture}\n\draw (0,0) -- (1,1);\n\end{tikzpicture}"
\end{verbatim}

\begin{verbatim}
(continues on next page)
\end{verbatim}
Utilities, Release 10.4

(continued from previous page)

```python
t = TikzPicture(s)
print(t.content())
\begin{tikzpicture}
draw (0,0) -- (1,1);
end{tikzpicture}
```

**dvi** (filename=None, view=True, program='latex')

Compile the latex code with latex and create a dvi file.

**INPUT:**

- filename – string (default: None), the output filename. If None, it saves the file in a temporary directory.
- view – bool (default: True), whether to open the file in a dvi viewer. This option is ignored and automatically set to False if filename is not None.
- program – string (default: 'latex'), 'latex'

**OUTPUT:**

string, path to dvi file

**EXAMPLES:**

```python
sage: from sage.misc.latex_standalone import Standalone
sage: t = Standalone('Hello World')
sage: _ = t.dvi(view=False)
# long time (1s)  # optional ~
←latex
```

```python
>>> from sage.all import *
```  ```python
>>> from sage.misc.latex_standalone import Standalone
```  ```python
>>> t = Standalone('Hello World')
```  ```python
>>> _ = t.dvi(view=False)
# long time (1s)  # optional ~
←latex
```

Same for instances of **TikzPicture:**

```python
sage: from sage.misc.latex_standalone import TikzPicture
sage: s = "\begin{tikzpicture}\n\draw (0,0) -- (1,1);\nend{tikzpicture}"
sage: t = TikzPicture(s)
sage: _ = t.dvi(view=False)
# not tested
```

```python
>>> from sage.all import *
```  ```python
>>> from sage.misc.latex_standalone import TikzPicture
```  ```python
>>> s = "\begin{tikzpicture}\n\draw (0,0) -- (1,1);\nend{tikzpicture}"
```  ```python
>>> t = TikzPicture(s)
```  ```python
>>> _ = t.dvi(view=False)
# not tested
```

A filename may be provided where to save the file, in which case the viewer does not open the file:

```python
sage: from sage.misc.temporary_file import tmp_filename
sage: filename = tmp_filename(temp,.dvi)
sage: path_to_file = t.dvi(filename)
# long time (1s)  # optional ~
←latex
```

```python
sage: path_to_file[-4:]
# long time (fast)  # optional ~
←latex
'.dvi'
```

498 Chapter 4. Formatted Output
The filename may contain spaces:

```python
sage: filename = tmp_filename('filename with spaces', '.dvi')
sage: path_to_file = t.dvi(filename)  # long time (1s)  # optional → latex
```
Utilities, Release 10.4

>>> t = TikzPicture(s)
>>> _ = t.eps(view=False)  # not tested

We test the creation of the files:

```
sage: from sage.misc.temporary_file import tmp_filename
sage: filename = tmp_filename('temp', '.eps')
```

```
sage: path_to_file = t.eps(filename,  
     # optional - latex dvips  
     ....:  
     # long time (1s)  
     # optional - latex dvips
sage: path_to_file[-4:]  
     # long time (fast)  
     # optional - latex dvips
'.eps'
sage: path_to_file = t.eps(filename,  
     # optional - latex pdftocairo
     ....:  
     program='pdftocairo')
```

```
sage: path_to_file[-4:]  
     # long time (fast)  
     # optional - latex pdftocairo
'.eps'
```

```
>>> from sage.all import *
>>> from sage.misc.temporary_file import tmp_filename
```

```
>>> filename = tmp_filename('temp', '.eps')
```

```
>>> path_to_file = t.eps(filename,  
     # long time (1s)  
     # optional - latex dvips
     ....:  
     program='dvips')
```

```
>>> path_to_file[-4:]  
     # long time (fast)  
     # optional - latex dvips
'.eps'
```

```
>>> path_to_file = t.eps(filename,  
     # optional - latex pdftocairo
     ....:  
     program='pdftocairo')
```

```
>>> path_to_file[-4:]  
     # long time (fast)  
     # optional - latex pdftocairo
'.eps'
```

```
pdf(filename=None, view=True, program=None)
```

Compile the latex code with pdflatex and create a pdf file.

INPUT:

- filename – string (default: None), the output filename. If None, it saves the file in a temporary directory.
- view – bool (default: True), whether to open the file in a pdf viewer. This option is ignored and automatically set to False if filename is not None.
- program – string (default: None) 'pdflatex' or 'lualatex'. If None, it uses 'lualatex' if it is available, otherwise 'pdflatex'.

OUTPUT:

string, path to pdf file

EXAMPLES:

```
sage: from sage.misc.latex_standalone import Standalone
sage: t = Standalone('Hello World')
```

(continues on next page)
sage: _ = t.pdf(view=False)  # long time (1s) # optional - latex

---

```python
>>> from sage.all import *
>>> from sage.misc.latex_standalone import Standalone
>>> t = Standalone('Hello World')
>>> _ = t.pdf(view=False)  # long time (1s) # optional - latex

```

Same for instances of `TikzPicture`:

```python
sage: from sage.misc.latex_standalone import TikzPicture
sage: s = r''
\begin{tikzpicture}
\draw (0,0) -- (1,1);
\end{tikzpicture}'

sage: t = TikzPicture(s)
sage: _ = t.pdf(view=False)  # not tested

---

```python
>>> from sage.all import *
>>> from sage.misc.latex_standalone import TikzPicture
>>> s = r''
\begin{tikzpicture}
\draw (0,0) -- (1,1);
\end{tikzpicture}'

>>> t = TikzPicture(s)
>>> _ = t.pdf(view=False)  # not tested

```

A filename may be provided where to save the file, in which case the viewer does not open the file:

```python
sage: from sage.misc.temporary_file import tmp_filename
sage: filename = tmp_filename(temp,.pdf)
sage: path_to_file = t.pdf(filename)  # long time (1s) # optional - latex

---

```python
>>> from sage.all import *
>>> filename = tmp_filename(filename with spaces,.pdf)
>>> path_to_file = t.pdf(filename)  # long time (1s) # optional - latex

```

The filename may contain spaces:

```python
sage: filename = tmp_filename('filename with spaces','.pdf')
sage: path_to_file = t.pdf(filename)  # long time (1s) # optional - latex

---

```python
>>> from sage.all import *
>>> filename = tmp_filename('filename with spaces','.pdf')
>>> path_to_file = t.pdf(filename)  # long time (1s) # optional - latex

```

```python
png (filename=None, density=150, view=True)
```

Compile the latex code with pdflatex and converts to a png file.

**INPUT:**
• **filename** – string (default: None), the output filename. If None, it saves the file in a temporary directory.
• **density** – integer, (default: 150), horizontal and vertical density of the image
• **view** – bool (default: True), whether to open the file in a png viewer. This option is ignored and automatically set to False if filename is not None.

**OUTPUT:**
string, path to png file

**EXAMPLES:**

```python
sage: from sage.misc.latex_standalone import Standalone
dsage: t = Standalone('Hello World')
dsage: _ = t.png(view=False) # long time (1s) # optional ~-
→latex imagemagick
```

```python
>>> from sage.all import *
>>> from sage.misc.latex_standalone import Standalone
>>> t = Standalone('Hello World')
>>> _ = t.png(view=False) # long time (1s) # optional - latex...
→imagemagick
```

Same for instances of `TikzPicture`:

```python
sage: from sage.misc.latex_standalone import TikzPicture
sage: s = "\begin{tikzpicture}\n\draw (0,0) -- (1,1);\n\end{tikzpicture}"
sage: t = TikzPicture(s)
sage: _ = t.png(view=False) # not tested
```

```python
>>> from sage.all import *
>>> from sage.misc.latex_standalone import TikzPicture
>>> s = "\begin{tikzpicture}\n\draw (0,0) -- (1,1);\n\end{tikzpicture}"
>>> t = TikzPicture(s)
>>> _ = t.png(view=False) # not tested
```

```python
sage: from sage.misc.temporary_file import tmp_filename
sage: filename = tmp_filename(temp,.png)
sage: path_to_file = t.png(filename) # long time (1s) # optional ~-
→latex imagemagick
```

```python
>>> from sage.all import *
>>> from sage.misc.temporary_file import tmp_filename
>>> filename = tmp_filename(temp,.png)
>>> path_to_file = t.png(filename) # long time (fast) # optional ~-
→latex imagemagick .png
```

```python
>>> from sage.all import *
>>> from sage.misc.temporary_file import tmp_filename
>>> filename = tmp_filename(temp,.png)
>>> path_to_file = t.png(filename) # long time (fast) # optional...
→latex imagemagick .png
```

**save (filename,** **kwds)**

Save the graphics to an image file.

**INPUT:**
• filename – string. The filename and the image format given by the extension, which can be one of the following:
  - .pdf,
  - .png,
  - .svg,
  - .eps,
  - .dvi,
  - .sobj (for a Sage object you can load later),
  - empty extension will be treated as .sobj.

All other keyword arguments will be passed to the plotter.

OUTPUT:
• None

Note: This method follows the signature of the method `sage.plot.Graphics.save()` in order to be compatible with with `sagetex`. In particular so that `\sageplot{t}` written in a tex file works when `t` is an instance of `Standalone` or `TikzPicture`.

EXAMPLES:

```python
sage: from sage.misc.temporary_file import tmp_filename
sage: from sage.misc.latex_standalone import Standalone
sage: t = Standalone('Hello World')
sage: filename = tmp_filename(temp,.pdf)
sage: t.save(filename)  # long time (1s)  
˓→optional - latex
sage: filename = tmp_filename(temp,.eps)
sage: t.save(filename)  # long time (1s)  
˓→optional - latex dvips
```

```bash
>>> from sage.all import *
>>> from sage.misc.temporary_file import tmp_filename
>>> from sage.misc.latex_standalone import Standalone

>>> t = Standalone('Hello World')

>>> filename = tmp_filename(temp,.pdf)

>>> t.save(filename)  # long time (1s)  
˓→latex

>>> filename = tmp_filename(temp,.eps)

>>> t.save(filename)  # long time (1s)  
˓→latex dvips
```

**svg** *(filename=None, view=True, program='pdftocairo')*

Compile the latex code with pdflatex and converts to a svg file.

INPUT:

• filename – string (default:None), the output filename. If None, it saves the file in a temporary directory.

• view – bool (default:True), whether to open the file in a browser. This option is ignored and automatically set to False if filename is not None.
Utilities, Release 10.4

• program – string (default:'pdftocairo') 'pdftocairo' or 'pdf2svg'.

OUTPUT:

string, path to svg file

EXAMPLES:

```
sage: from sage.misc.latex_standalone import Standalone
sage: t = Standalone('Hello World')
sage: _ = t.svg(view=False)  # not tested

>>> from sage.all import *
>>> from sage.misc.latex_standalone import Standalone
>>> t = Standalone('Hello World')
>>> _ = t.svg(view=False)  # not tested
```

Same for instances of `TikzPicture`:

```
sage: from sage.misc.latex_standalone import TikzPicture
sage: s = r'\begin{tikzpicture}
\begin{axis}
\end{axis}
\end{tikzpicture}'
sage: t = TikzPicture(s)
sage: _ = t.svg(view=False)  # not tested

>>> from sage.all import *
>>> from sage.misc.latex_standalone import TikzPicture
>>> s = r'\begin{tikzpicture}
\begin{axis}
\end{axis}
\end{tikzpicture}'
>>> t = TikzPicture(s)
>>> _ = t.svg(view=False)  # not tested
```

```
sage: from sage.misc.temporary_file import tmp_filename
sage: filename = tmp_filename(temp, '.svg')
sage: path_to_file = t.svg(filename,  # long time (1s)  
                      # optional - latex pdf2svg  
                      ...)
                      # program='pdf2svg')
                      # long time (fast)  
                      # optional - latex pdf2svg  
                      
                      path_to_file[-4:]  # long time (fast)
                      # optional - latex pdftocairo  
                      
                      path_to_file = t.svg(filename,  # long time (1s)  
                      # optional - latex pdf2svg  
                      ...  
                      # program='pdftocairo')
                      # long time (fast)
                      # optional - latex pdftocairo  
                      'svg'

>>> from sage.all import *
>>> from sage.misc.temporary_file import tmp_filename
>>> filename = tmp_filename(temp, '.svg')
>>> path_to_file = t.svg(filename,  # long time (1s)  
                       # optional - latex pdf2svg  
                       ...  
                       # program='pdf2svg')
>>> path_to_file[-Integer(4):]  # long time (fast)  
                       # optional - latex pdf2svg  
                       
                       path_to_file = t.svg(filename,  # long time (1s)  
                       # optional - latex pdf2cairo  
                       ...  
                       # program='pdftocairo')
```

(continues on next page)
tex (filename=None, content_only=False, include_header=None)

Writes the LaTeX code to a file.

INPUT:

- filename – string (default: None), the output filename. If None, it saves the file in a temporary directory.
- content_only – bool (default: False) whether to include the header LaTeX part. If True, it prints only the content to the file.

OUTPUT:

string, path to tex file

EXAMPLES:

```python
sage: from sage.misc.latex_standalone import Standalone
sage: t = Standalone('Hello World')
sage: _ = t.tex()
sage: _ = t.tex(content_only=True)
```

Same for instances of TikzPicture:

```python
sage: from sage.misc.latex_standalone import TikzPicture
sage: s = r'\begin{tikzpicture}
\draw (0,0) -- (1,1);
\end{tikzpicture}'
sage: t = TikzPicture(s)
sage: _ = t.tex()
sage: _ = t.tex(content_only=True)
```

Write to a given filename:

```python
sage: from sage.misc.temporary_file import tmp_filename
sage: filename = tmp_filename('.tex')
sage: path_to_file = t.tex(filename)
sage: path_to_file[-4:]
'.svg'
```

(continues on previous page)
class sage.misc.latex_standalone.TikzPicture(content, standalone_config=None, usepackage=None,usetikzlibrary=None, macros=None, use_sage_preamble=False)

Bases: Standalone

A TikzPicture embedded in a LaTeX standalone document class.

INPUT:

• content – string, tikzpicture code starting with r'\begin{tikzpicture}' and ending with r'\end{tikzpicture}'
• standalone_config – list of strings (default: []), latex document class standalone configuration options.
• usepackage – list of strings (default: []), latex packages.
• usetikzlibrary – list of strings (default: []), tikz libraries to use.
• macros – list of strings (default: []), stuff you need for the picture.
• use_sage_preamble – bool (default: False), whether to include sage latex preamble and sage latex macros, that is, the content of sage.misc.latex.extra_preamble(), sage.misc.latex.extra_macros() and sage.misc.latex_macros.sage_latex_macros().

EXAMPLES:

Create your own tikz string from scratch and provide it:

```python
>>> from sage.misc.latex_standalone import TikzPicture
>>> lines = []
>>> lines.append(r'\begin{tikzpicture}
>>> lines.append(r'\draw[very thick,orange,->] (0,0) -- (1,1);')
>>> lines.append(r'\end{tikzpicture}'))
>>> s = '\n'.join(lines)
>>> t = TikzPicture(s)
```

```latex
\documentclass[tikz]{standalone}
\begin{document}
\begin{tikzpicture}
\draw[very thick,orange,->] (0,0) -- (1,1);
\end{tikzpicture}
\end{document}
```
Then use it by exporting the tikzpicture to other formats, all of the below methods return a string providing the path to the filename, which is by default in a temporary folder:

```python
sage: # not tested
sage: _ = t.pdf()
sage: _ = t.png()
sage: _ = t.svg()
sage: _ = t.tex()
sage: _ = t.pdf(filename='abc.pdf')
```

Here we create a tikzpicture for the latex representation of a graph. This is using tkz-graph tex library:

```python
sage: # needs sage.graphs
sage: g = graphs.PetersenGraph()
sage: s = latex(g) # optional - latex
sage: t = TikzPicture(s, standalone_config=['border=4mm'], # optional - latex
...: usepackage=['tkz-graph'])
sage: _ = t.pdf(view=False) # long time (2s), optional - latex
```

Here are standalone configurations, packages, tikz libraries and macros that can be set:

```python
sage: options = ['preview', 'border=4mm', 'beamer', 'float']
sage: usepackage = ['nicefrac', 'amsmath', 'pifont', 'tikz-3dplot',
...: 'pgfplots']
sage: tikzlib = ['arrows', 'snakes', 'backgrounds', 'patterns',
...: 'matrix', 'shapes', 'fit', 'calc', 'shadows', 'plotmarks',
...: 'positioning', 'pgfplots.groupplots', 'mindmap']
sage: macros = ['\newcommand{\ZZ}{\mathbb{Z}}']
sage: s = '\begin{tikzpicture}\draw[very thick,orange,-] (0,0) -- (1,1);\end{tikzpicture}'
sage: t = TikzPicture(s, standalone_config=options, usepackage=usepackage,
...: tikzlib=tikzlib, macros=macros)
sage: _ = t.pdf(view=False) # long time (2s), optional - latex
```
add_usetikzlibrary (library)

Add ausetikzlibrary line

INPUT:

• library – string, name of library

EXAMPLES:

```python
>>> from sage.misc.latex_standalone import TikzPicture

>>> s = r'\begin{tikzpicture}\n\draw (0,0) -- (1,1);\n\end{tikzpicture}'

>>> t = TikzPicture(s, standalone_config=options, usepackage=usepackage,
... useusetikzlibrary=tikzlib, macros=macros)

>>> _ = t.pdf(view=False)  # long time (2s), optional - latex
```

```
\documentclass[tikz]{standalone}
\usetikzlibrary{arrows}
\begin{document}
\begin{tikzpicture}
\draw (0,0) -- (1,1);
\end{tikzpicture}
\end{document}
```
Utilities, Release 10.4

- **prog** — string (default: 'dot') the program used for the layout corresponding to one of the software of the graphviz suite: ‘dot’, ‘neato’, ‘twopi’, ‘circo’ or ‘fdp’.

**EXAMPLES:**

```python
sage: # needs sage.graphs
cython: from sage.misc.latex_standalone import TikzPicture
cython: G = graphs.PetersenGraph()
cython: dotdata = G.graphviz_string()
cython: tikz = TikzPicture.from_dot_string(dotdata)  # long time (3s),
cython: # optional - dot2tex graphviz
sage: _ = tikz.pdf()  # not tested
```

```python
>>> from sage.all import *
>>> # needs sage.graphs
>>> cython: from sage.misc.latex_standalone import TikzPicture
cython: G = graphs.PetersenGraph()
cython: dotdata = G.graphviz_string()
cython: tikz = TikzPicture.from_dot_string(dotdata)  # long time (3s), optional -
cython: # dot2tex graphviz
sage: _ = tikz.pdf()  # not tested
```

```python
sage: # needs sage.graphs
cython: dotdata = G.graphviz_string(labels='latex')
cython: tikz = TikzPicture.from_dot_string(dotdata)  # long time (3s),
cython: # optional - dot2tex graphviz
sage: _ = tikz.pdf()  # not tested
```

```python
>>> from sage.all import *
>>> # needs sage.graphs
>>> dotdata = G.graphviz_string(labels='latex')
cython: tikz = TikzPicture.from_dot_string(dotdata)  # long time (3s), optional -
cython: # dot2tex graphviz
sage: _ = tikz.pdf()  # not tested
```

```python
sage: # needs sage.combinat sage.graphs sage.groups
cython: W = CoxeterGroup(['A',2])
cython: G = W.cayley_graph()
cython: dotdata = G.graphviz_string()
cython: tikz = TikzPicture.from_dot_string(dotdata)  # long time (3s),
cython: # optional - dot2tex graphviz
sage: _ = tikz.pdf()  # not tested
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.graphs sage.groups
cython: W = CoxeterGroup(['A',Integer(2)])
cython: G = W.cayley_graph()
cython: dotdata = G.graphviz_string()
cython: tikz = TikzPicture.from_dot_string(dotdata)  # long time (3s), optional -
cython: # dot2tex graphviz
sage: _ = tikz.pdf()  # not tested
```

```python
sage: # needs sage.combinat sage.graphs sage.groups
cython: dotdata = G.graphviz_string(labels='latex')
cython: tikz = TikzPicture.from_dot_string(dotdata)  # long time (3s),
cython: # optional - dot2tex graphviz
```

(continues on next page)
classmethod from_graph(graph, merge_multiedges=True, merge_label_function=<class 'tuple'>, **kwds)

Convert a graph to a tikzpicture using graphviz and dot2tex.

Note: Prerequisite: dot2tex optional Sage package and graphviz must be installed.

Warning: This method might be deleted in the future in favor of a method in the graph class returning a tikz picture.

INPUT:

- graph – graph
- merge_multiedges – bool (default: True), if the graph has multiple edges, whether to merge the multiedges into one single edge
- merge_label_function – function (default: tuple), a function to apply to each list of labels to be merged. It is ignored if merge_multiedges is not True or if the graph has no multiple edges.

Other inputs are used for latex drawing with dot2tex and graphviz:

- prog – string (default: 'dot') the program used for the layout corresponding to one of the software of the graphviz suite: ‘dot’, ‘neato’, ‘twopi’, ‘circo’ or ‘fdp’.
- edge_labels – bool (default: True)
- color_by_label – bool (default: False)
- rankdir – string (default: 'down')
- subgraph_clusters – (default: []) a list of lists of vertices, if supported by the layout engine, nodes belonging to the same cluster subgraph are drawn together, with the entire drawing of the cluster contained within a bounding rectangle.

EXAMPLES:
Utilities, Release 10.4

```python
>>> from sage.all import *
>>> # needs sage.graphs
>>> from sage.misc.latex_standalone import TikzPicture
>>> g = graphs.PetersenGraph()
>>> tikz = TikzPicture.from_graph(g)   # optional - dot2tex graphviz
```

---

```python
...: FutureWarning: This class/method/function is marked as experimental.
It, its functionality or its interface might change without a formal deprecation.
See https://github.com/sagemath/sage/issues/20343 for details.

>>> _ = tikz.pdf() # not tested
```

Using `prog`:

```python
sage: # needs sage.graphs
tikz = TikzPicture.from_graph(g, prog='neato', ... # long time (3s),
    optional - dot2tex graphviz
    color_by_label=True)
```

```python
>> from sage.all import *
>>> # needs sage.graphs
>>> tikz = TikzPicture.from_graph(g, prog='neato', ... # long time (3s),
    optional - dot2tex graphviz, color_by_label=True)
```

Using `rankdir`:

```python
sage: tikz = TikzPicture.from_graph(g, rankdir='right')   # optional - dot2tex graphviz, needs sage.graphs
```

```python
>>> from sage.all import *
>>> tikz = TikzPicture.from_graph(g, rankdir='right')   # optional - dot2tex graphviz, needs sage.graphs
```

Using `merge_multiedges`:

```python
sage: # needs sage.graphs sage.modules sage.symbolic
sage: alpha = var('alpha')
sage: m = matrix(2, range(4)); m.set_immutable()
sage: G = DiGraph([(0,1,alpha), (0,1,0), (0,2,9), (0,2,m)], multiedges=True)
```

```python
sage: tikz = TikzPicture.from_graph(G, merge_multiedges=True)   # ...
```

```python
>>> from sage.all import *
>>> # needs sage.graphs sage.modules sage.symbolic
>>> alpha = var('alpha')
>>> m = matrix(Integer(2), range(Integer(4))); m.set_immutable()
>>> G = DiGraph([(Integer(0),Integer(1),alpha), (Integer(0),Integer(0), (Integer(2),m)], ...
```

(continues on next page)
... multiedges=True)
>>> tikz = TikzPicture.from_graph(G, merge_multiedges=True)        # optional -
← dot2tex graphviz
>>> _ = tikz.pdf()         # not tested

Using merge_multiedges with merge_label_function:

sage: # needs sage.graphs
sage: fn = lambda L: LatexExpr(','.join(map(str, L)))
sage: edges = [(0,1,'a'), (0,1,'b'), (0,2,'c'), (0,2,'d')]
sage: G = DiGraph(edges, multiedges=True)

# optional - dot2tex graphviz
...

Using subgraphs clusters (broken when using labels, see Issue #22070):

sage: S = FiniteSetMaps(5)
sage: I = S((0,1,2,3,4))
sage: a = S((0,1,3,0,0))
sage: b = S((0,2,4,1,0))

sage: roots = [I]

sage: succ = lambda v: [v*a,v*b,a*v,b*v]

sage: R = RecursivelyEnumeratedSet(roots, succ)

sage: G = R.to_digraph()        # needs sage.graphs

sage: C = G.strongly_connected_components()        # needs sage.graphs

sage: tikz = TikzPicture.from_graph(G,
# optional - dot2tex graphviz
...

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

>>> G = R.to_digraph()  # needs sage.graphs
>>> G  # needs sage.graphs
Looped multi-digraph on 27 vertices
>>> C = G.strongly_connected_components()  # needs sage.graphs
>>> tikz = TikzPicture.from_graph(G,  # optional - dot2tex graphviz, needs sage.graphs
...     merge_multiedges=False, subgraph_clusters=C)
>>> _ = tikz.pdf()  # not tested

An example coming from graphviz_string documentation in SageMath:

```
sage: # needs sage.graphs sage.symbolic
sage: f(x) = -1 / x
sage: g(x) = 1 / (x + 1)
sage: G = DiGraph()
sage: G.add_edges((i, f(i), f) for i in (1, 2, 1/2, 1/4))
sage: G.add_edges((i, g(i), g) for i in (1, 2, 1/2, 1/4))
sage: tikz = TikzPicture.from_graph(G)  # optional - dot2tex graphviz
sage: _ = tikz.pdf()  # not tested
```

```
sage: def edge_options(data):
...     u, v, label = data
...     options = {"color": {f: "red", g: "blue"}[label]}
...     if (u,v) == (1/2, -2): options["label"] = "coucou"; options[
\"label_style\"] = "string"
...     if (u,v) == (1/2,2/3): options["dot"] = "x=1,y=2"
...     if (u,v) == (1, -1): options["label_style"] = "latex"
...     if (u,v) == (1, 1/2): options["dir"] = "back"
...     return options
sage: tikz = TikzPicture.from_graph(G, edge_options=edge_options)  #
\optional - dot2tex graphviz
sage: _ = tikz.pdf()  # not tested
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.symbolic
>>> __tmp__=var("x"); f = symbolic_expression(-Integer(1) / x).function(x)
>>> __tmp__=var("x"); g = symbolic_expression(Integer(1) / (x + Integer(1))).function(x)
>>> G = DiGraph()
>>> G.add_edges((i, f(i), f) for i in (Integer(1), Integer(2), Integer(1)/Integer(2), Integer(1)/Integer(4)))
>>> G.add_edges((i, g(i), g) for i in (Integer(1), Integer(2), Integer(1)/Integer(2), Integer(1)/Integer(4)))
>>> tikz = TikzPicture.from_graph(G)  # optional - dot2tex graphviz
>>> _ = tikz.pdf()  # not tested
```

(continues on next page)
if (u,v) == (Integer(1), Integer(1)/Integer(2)): options["dir"] = "back"

return options

>>> tikz = TikzPicture.from_graph(G, edge_options=edge_options)  # optional -

>>> _ = tikz.pdf()  # not tested

classmethod from_graph_with_pos

Convert a graph with positions defined for vertices to a tikzpicture.

**Warning**: This method might be deleted in the future in favor of a method in the graph class returning a tikz picture.

**INPUT:**

- graph – graph (with predefined positions)
- scale – number (default: 1), tikzpicture scale
- merge_multiedges – bool (default: True), if the graph has multiple edges, whether to merge the multiedges into one single edge
- merge_label_function – function (default: tuple), a function to apply to each list of labels to be merged. It is ignored if merge_multiedges is not True or if the graph has no multiple edges.

**EXAMPLES:**

```python
def example_graph():
    edges = [(0,0,'a'),(0,1,'b'),(0,1,'c')]
    kwds = dict(format='list_of_edges', loops=True, multiedges=True)
    G = DiGraph(edges, **kwds)
    return G

def main():
    G = example_graph()
    tikz = TikzPicture.from_graph_with_pos(G, edge_options=edge_options)
    tikz.pdf()  # not tested
```
```python
sage: G.set_pos({0:(0,0), 1:(1,0)})
sage: f = lambda label: ','.join(label)
sage: TikzPicture.from_graph_with_pos(G, merge_label_function=f)
```

```latex
\documentclass[tikz]{standalone}
\standaloneconfig{border=4mm}
\begin{document}
\begin{tikzpicture}
[auto,scale=1]
% vertices
\node (node_0) at (0, 0) {0};
\node (node_1) at (1, 0) {1};
% edges
\draw[->] (node_0) -- node {b,c} (node_1);
% loops
\draw (node_0) edge [loop above] node {a} (node_0);
\end{tikzpicture}
\end{document}
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> edges = [(Integer(0),Integer(0),a),(Integer(0),Integer(1),b), ...
\rightarrow (Integer(0),Integer(1),c)]
>>> kwds = dict(format=list_of_edges, loops=True, multiedges=True)
>>> G = DiGraph(edges, **kwds)
>>> G.set_pos({Integer(0):(Integer(0),Integer(0)), Integer(1):(Integer(1), ...
\rightarrow Integer(0))})
>>> f = lambda label: ','.join(label)
>>> TikzPicture.from_graph_with_pos(G, merge_label_function=f)
```

```latex
\documentclass[tikz]{standalone}
\standaloneconfig{border=4mm}
\begin{document}
\begin{tikzpicture}
[auto,scale=1]
% vertices
\node (node_0) at (0, 0) {0};
\node (node_1) at (1, 0) {1};
% edges
\draw[->] (node_0) -- node {b,c} (node_1);
% loops
\draw (node_0) edge [loop above] node {a} (node_0);
\end{tikzpicture}
\end{document}
```

`classmethod from_poset(poset, **kwds)`

Convert a poset to a `tikzpicture` using graphviz and dot2tex.

**Note:** Prerequisite: `dot2tex` optional Sage package and `graphviz` must be installed.

**Warning:** This method might be deleted in the future in favor of a method in the `graph` class returning a `tikz` picture.

**INPUT:**

4.3. LaTeX 515
• poset – poset
• prog – string (default: 'dot') the program used for the layout corresponding to one of the software of the graphviz suite: 'dot', 'neato', 'twopi', 'circo' or 'fdp'.
• edge_labels – bool (default: True)
• color_by_label – bool (default: False)
• rankdir – string (default: 'down')

EXAMPLES:

```python
sage: from sage.misc.latex_standalone import TikzPicture
```

```python
sage: # needs sage.graphs sage.modules
sage: P = posets.PentagonPoset()
```

```python
doctest:...: FutureWarning: This class/method/function is marked as experimental.
It, its functionality or its interface might change without a formal deprecation.
See https://github.com/sagemath/sage/issues/20343 for details.
```

```python
>>> from sage.all import *
```

```python
>>> from sage.misc.latex_standalone import TikzPicture
```

```python
>>> # needs sage.graphs sage.modules
>>> P = posets.PentagonPoset()
```

```python
>>> tikz = TikzPicture.from_poset(P)  # optional - dot2tex graphviz
doctest:...: FutureWarning: This class/method/function is marked as experimental.
It, its functionality or its interface might change without a formal deprecation.
See https://github.com/sagemath/sage/issues/20343 for details.
```

```python
sage: tikz = TikzPicture.from_poset(P, prog=neato, # long time (3s),
                                      # optional - dot2tex, needs sage.graphs sage.modules
                                      color_by_label=True)
```

```python
>>> from sage.all import *
```

```python
>>> from sage.misc.latex_standalone import TikzPicture
```

```python
>>> # needs sage.graphs sage.modules
>>> P = posets.SymmetricGroupWeakOrderPoset(4)
```

```python
>>> tikz = TikzPicture.from_poset(P)
```

```python
>>> tikz = TikzPicture.from_poset(P, prog='neato',  # long time (4s),
                                      # optional - dot2tex graphviz
                                      color_by_label=True)
```

```python
sage: # needs sage.graphs
sage: P = posets.SymmetricGroupWeakOrderPoset(4)
```

```python
sage: tikz = TikzPicture.from_poset(P, prog='neato')  # long time (4s),
```

```python
>>> from sage.all import *
```

```python
>>> P = posets.SymmetricGroupWeakOrderPoset(Integer(4))
```

```python
>>> tikz = TikzPicture.from_poset(P)  # long time (4s),
```

(continues on next page)
tikz_picture_code()

EXAMPLES:

```python
sage: from sage.misc.latex_standalone import TikzPicture
sage: s = "\begin{tikzpicture}\\draw (0,0) -- (1,1);\\end{tikzpicture}"

sage: t = TikzPicture(s)

sage: print(t.tikz_picture_code())
\begin{tikzpicture}
\draw (0,0) -- (1,1);
\end{tikzpicture}
```

```python
>>> from sage.all import *

>>> from sage.misc.latex_standalone import TikzPicture

>>> s = "\begin{tikzpicture}\\draw (0,0) -- (1,1);\\end{tikzpicture}"

>>> t = TikzPicture(s)

>>> print(t.tikz_picture_code())
\begin{tikzpicture}
\draw (0,0) -- (1,1);
\end{tikzpicture}
```
5.1 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,

```python
@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.

```python
class sage.misc.func_persist.func_persist(f, dir='func_persist')
Bases: object

    Put @func_persist right before your function definition to cache values it computes to disk.
```

5.2 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 \( x \) 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.
class \texttt{sage.misc.persist.SagePickler}(\texttt{file_obj}, \texttt{persistent_id=None}, \texttt{py2compat=True})

Bases: \texttt{_BasePickler}

Subclass \texttt{pickle.Pickler} with Sage-specific default options, and built-in support for external object persistence.

INPUT:

- \texttt{file_obj} – a readable file-like object returning \texttt{bytes} from which the pickle data will be loaded.
- \texttt{persistent_id} – callable or None; if given this callable takes a single object to be pickled, and returns an “ID” (a key with which to restore the object upon unpickling, which may itself be any pickleable object). See the Python documentation on pickling and unpickling external objects for more details.
- \texttt{py2compat} – on Python 3 only, this creates pickles that have a better chance of being read on Python 2, by using protocol version 2 (instead of 4) and fixing up imports of standard library modules and types whose names changed between Python 2 and 3. This is enabled by default for the best chances of cross-Python compatibility.
- Further arguments are passed to \texttt{pickle.load()}, where in Python-3 Sage sets the default \texttt{encoding='latin1'}. This is essential to make pickles readable in Python-3 that were created in Python-2. See Issue \#28444 for details.

EXAMPLES:

```python
sage: from sage.misc.persist import ...
    unpickler_override, register_unpickler_override, SageUnpickler)

sage: from sage.misc.persist import ...
    unpickler_override, register_unpickler_override, SageUnpickler)

sage: from sage.rings.integer import make_integer

sage: from io import BytesIO

sage: def fake_constructor(x):
    ...
    print("unpickling an Integer")
    ...
    return make_integer(x)

sage: register_unpickler_override('sage.rings.integer', 'make_integer',
    ...
    fake_constructor)

sage: unp = SageUnpickler(BytesIO(dumps(1, compress=False)))

sage: unp.load()
unpickling an Integer
1

sage: del unpickler_override[('sage.rings.integer', 'make_integer')]
```

```bash
>>> from sage.all import *

>>> from sage.misc.persist import ...
    unpickler_override, register_unpickler_override, SageUnpickler)

>>> from sage.rings.integer import make_integer

>>> from io import BytesIO

>>> def fake_constructor(x):
    ...
    print("unpickling an Integer")
    ...
    return make_integer(x)

>>> register_unpickler_override('sage.rings.integer', 'make_integer',
    ...
    fake_constructor)

>>> unp = SageUnpickler(BytesIO(dumps(Integer(1), compress=False)))

>>> unp.load()
unpickling an Integer
1

>>> del unpickler_override[('sage.rings.integer', 'make_integer')]
```

The \texttt{SagePickler} can also be passed a \texttt{persistent_id} function:
```python
table = {1: 'a', 2: 'b'}
def load_object_from_table(obj_id):
    return table[obj_id]

table = {Integer(1): 'a', Integer(2): 'b'}
def load_object_from_table(obj_id):
    return table[obj_id]
```

```python
classmethod dumps(obj, **kwargs)

Equivalent to pickle.dumps() but using the sage.misc.persist.SagePickler.

INPUT:
- obj – the object to pickle.
- kwargs – keyword arguments passed to the sage.misc.persist.SagePickler constructor.

OUTPUT:
- pickle – the pickled object as bytes.

EXAMPLES:
```
sage: import pickle
sage: from sage.misc.persist import SagePickler
sage: gherkin = SagePickler.dumps(1)
sage: pickle.loads(gherkin)
1
```

class sage.misc.persist.SageUnpickler(file_obj, persistent_load=None, **kwargs)

Bases: _BaseUnpickler

Subclass pickle.Unpickler to control how certain objects get unpickled (registered overrides, specifically).

This is only needed in Python 3 and up. On Python 2 the behavior of the cPickle module is customized differently.

This class simply overrides Unpickler.find_class to wrap sage.misc.persist.unpickle_global.

INPUT:
- file_obj – a readable file-like object returning bytes from which the pickle data will be loaded.
- persistent_load – callable or None; if given this callable implements loading of persistent external objects. The function should take a single argument, the persistent object ID. See the Python documentation on pickling and unpickling external objects for more details.
- kwargs – additional keyword arguments passed to the pickle.Unpickler constructor.
EXAMPLES:

```python
sage: from sage.misc.persist import (unpickle_override, register_unpickle_override, SageUnpickler)
sage: from sage.rings.integer import make_integer
sage: from io import BytesIO
sage: def fake_constructor(x):
....:     print("unpickling an Integer")
....:     return make_integer(x)
sage: register_unpickle_override(\'sage.rings.integer\', \'make_integer\', fake_constructor)
sage: unp = SageUnpickler(BytesIO(dumps(1, compress=False)))))
sage: unp.load()
unpickling an Integer
1
sage: del unpickle_override[\'sage.rings.integer\', \'make_integer\']
```

The `SageUnpickler` can also be passed a persistent_load function:

```python
sage: table = {1: a, 2: b}
sage: # in practice this might be a database or something...
sage: def load_object_from_table(obj_id):
....:     tag, obj_id = table[obj_id]
....:     return tag
>>> from sage.all import *
>>> from sage.misc.persist import (unpickle_override, register_unpickle_override, SageUnpickler)
>>> from sage.rings.integer import make_integer
>>> from io import BytesIO
>>> def fake_constructor(x):
...     print("unpickling an Integer")
...     return make_integer(x)
>>> register_unpickle_override(\'sage.rings.integer\', \'make_integer\', fake_constructor)
>>> unp = SageUnpickler(BytesIO(dumps(Integer(1), compress=False)))))
>>> unp.load()
unpickling an Integer
1
>>> del unpickle_override[\'sage.rings.integer\', \'make_integer\']
```

classmethod `loads`(data, **kwargs)

Equivalent to `pickle.loads()` but using the `sage.misc.persist.SagePickler`

INPUT:

- data – the pickle data as bytes.
- kwargs – keyword arguments passed to the `sage.misc.persist.SageUnpickler` constructor.

OUTPUT:
Utilities, Release 10.4

- `obj` – the object that was serialized to the given pickle data.

EXAMPLES:

```python
sage: import pickle
sage: from sage.misc.persist import SageUnpickler
sage: gherkin = pickle.dumps(1)
sage: SageUnpickler.loads(gherkin)
1
```

```python
>>> from sage.all import *
>>> import pickle
>>> from sage.misc.persist import SageUnpickler
>>> gherkin = pickle.dumps(Integer(1))
>>> SageUnpickler.loads(gherkin)
1
```

`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.dumps(obj, compress=True)`

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

See also:

`loads()`

EXAMPLES:

```python
sage: a = 2/3
sage: s = dumps(a)
sage: a2 = loads(s)
sage: type(a) is type(a2)
True
sage: a2
2/3
```

```python
>>> from sage.all import *
>>> a = Integer(2)/Integer(3)
>>> s = dumps(a)
>>> a2 = loads(s)
>>> type(a) is type(a2)
True
>>> a2
2/3
```

`sage.misc.persist.load(compress=True, verbose=True, *filename, **kwargs)`

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

5.2. Object persistence 523
Loaded files are not loaded into their own namespace, i.e., this is much more like Python’s `execfile` than Python’s `import`.

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

When a pickle created with Python 2 is unpickled in Python 3, Sage uses the default encoding `latin1` to unpickle data of type `str`.

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

If `compress` is true (the default), then the data stored in the file are supposed to be compressed. If `verbose` is true (the default), then some logging is printed when accessing remote files. Further keyword arguments are passed to `pickle.load()`.

**EXAMPLES:**

```python
sage: u = 'https://www.sagemath.org/files/test.sobj'  # optional

Attempting to load remote file: https://www.sagemath.org/files/test.sobj
Loading started
Loading ended
sage: s
'hello SageMath'
```

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

```python
sage: t = tmp_filename(ext='.py')

sage: with open(t, 'w') as f:
....:     _ = f.write("print('hello world')")

sage: load(t)
hello world
sage: load(t,t)
hello world
hello world

sage: t2 = tmp_filename(); save(2/3,t2)

sage: load(t,t,t2)
hello world
hello world

[None, None, 2/3]
```

```python
>>> from sage.all import *

>>> t = tmp_filename(ext='.py')

```
... _ = f.write("print('hello world')")
>>> load(t)
hello world
>>> load(t,t)
hello world
hello world
>>> t2 = tmp_filename(); save(Integer(2)/Integer(3),t2)
>>> load(t,t,t2)
hello world
hello world
[None, None, 2/3]

Files with a .sage extension are preparsed. Also note that we can access global variables:

```python
sage: t = tmp_filename(ext=".sage")
sage: with open(t, 'w') as f:
    ....: _ = f.write("a += Mod(2/3, 11)"")  # This evaluates to Mod(8, 11)
sage: a = -1
sage: load(t)
sage: a
7
```

```python
>>> from sage.all import *
>>> t = tmp_filename(ext=".sage")
>>> with open(t, 'w') as f:
    ... _ = f.write("a += Mod(2/3, 11)"")  # This evaluates to Mod(8, 11)
>>> a = ~Integer(1)
>>> load(t)
>>> a
7
```

We can load Fortran files:

```python
sage: code = 'subroutine hello
print *, "Hello World!"
end
'
sage: t = tmp_filename(ext=".F")
sage: with open(t, 'w') as f:
    ....: _ = f.write(code)
sage: load(t)
# needs numpy
sage: hello
# needs numpy
<fortran ...>
```

```python
>>> from sage.all import *
>>> code = 'subroutine hello
print *, "Hello World!"
end
'
>>> t = tmp_filename(ext=".F")
>>> with open(t, 'w') as f:
    ... _ = f.write(code)
>>> load(t)
# needs numpy
>>> hello
# needs numpy
<fortran ...>
```

`sage.misc.persist.load_sage_element(cls, parent, dic_pic)`
sage.misc.persist.load_sage_object(cls, dic)

sage.misc.persist.loads(s, compress=True, **kwargs)

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

See also:
dumps()

EXAMPLES:

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

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

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

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

The next example demonstrates that Sage strives to avoid data loss in the transition from Python-2 to Python-3. The problem is that Python-3 by default would not be able to unpickle a non-ASCII Python-2 string appearing in
a pickle. See Issue #28444 for details.

```python
sage: class Foo():
    ....:     def __init__(self, s):
    ....:         self.bar = s
    ....:     def __reduce__(self):
    ....:         return Foo, (self.bar,)

sage: import __main__
sage: __main__.Foo = Foo
```

```python
>>> from sage.all import *
```  

```python
>>> class Foo():
...     def __init__(self, s):
...         self.bar = s
...     def __reduce__(self):
...         return Foo, (self.bar,)

>>> import __main__
>>> __main__.Foo = Foo
```

The data that is passed to `loads` in the following line was created by `dumps(Foo(\x80\x07))` in Python-2.

```python
sage: g = loads(b'\x9c\x8e\x8f\xc8\xc9\x8b\x8f\xe7\xce\xcf\xe7*d\x0e\x8f')
```

```python
sage: type(g), g.bar
(<class '__main__.Foo'>, '\x80\x07')
```

```python
>>> from sage.all import *
```  

```python
>>> g = loads(b'\x9c\x8e\x8f\xc8\xc9\x8b\x8f\xe7\xce\xcf\xe7*d\x0e\x8f')
```

```python
>>> type(g), g.bar
(<class '__main__.Foo'>, '\x80\x07')
```

The following line demonstrates what would happen without Issue #28444:

```python
sage: loads(b'\x9c\x8e\x8f\xc8\xc9\x8b\x8f\xe7\xce\xcf\xe7*d\x0e\x8f')
```

```
Traceback (most recent call last):
  ...  
UnicodeDecodeError: 'ascii' codec can't decode byte 0x80 in position 0: ordinal not in range(128)
```

```python
>>> from sage.all import *
```  

```python
>>> loads(b'\x9c\x8e\x8f\xc8\xc9\x8b\x8f\xe7\xce\xcf\xe7*d\x0e\x8f')
```

```
Traceback (most recent call last):
  ...  
UnicodeDecodeError: 'ascii' codec can't decode byte 0x80 in position 0: ordinal not in range(128)
```

`sage.misc.persist.make_None(*args, **kwds)`

Do nothing and return None. Used for overriding pickles when that pickle is no longer needed.

**EXAMPLES:**
Utilities, Release 10.4

```
sage: from sage.misc.persist import make_None
sage: print(make_None(42, pi, foo='bar'))  # needs sage.symbolic
None

>>> from sage.all import *
>>> from sage.misc.persist import make_None
>>> print(make_None(Integer(42), pi, foo='bar'))  # needs sage.symbolic
None
```

`sage.misc.persist.picklejar(obj, dir=None)`

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

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

**INPUT:**

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

**EXAMPLES:**

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

```
>>> from sage.all import *
>>> dir = tmp_dir()
>>> sage.misc.persist.picklejar(Integer(1), dir)
>>> sage.misc.persist.picklejar('test', dir)
>>> len(os.listdir(dir))  # Two entries (sobj and txt) for each object
4
```

`sage.misc.persist.register_unpickle_override(module, name, callable, call_name=None)`

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

**EXAMPLES:**

Imagine that there used to be an `old_integer` module and old pickles essentially trying to do the following:

```
sage: unpickle_global('sage.rings.old_integer', 'OldInteger')
Traceback (most recent call last):
...  
ImportError: cannot import OldInteger from sage.rings.old_integer, call register_
    unpickle_override('sage.rings.old_integer', 'OldInteger', ...) to fix this
```
Utilities, Release 10.4

>>> from sage.all import *
>>> unpickle_global('sage.rings.old_integer', 'OldInteger')
Traceback (most recent call last):
... ImportError: cannot import OldInteger from sage.rings.old_integer, call register_unpickle_override('sage.rings.old_integer', 'OldInteger', ...) to fix this

After following the advice from the error message, unpickling works:

```
sage: from sage.misc.persist import register_unpickle_override
sage: register_unpickle_override('sage.rings.old_integer', 'OldInteger', Integer)
sage: unpickle_global('sage.rings.old_integer', 'OldInteger')
<... sage.rings.integer.Integer>
```

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

```
object.__setstate__(state)

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

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

```
sage: from sage.structure.element import Element
sage: class SourPickle(CombinatorialObject): pass
sage: class SweetPickle(CombinatorialObject, Element): pass
sage: import __main__
sage: __main__.SourPickle = SourPickle
sage: __main__.SweetPickle = SweetPickle # a hack to allow us to pickle command_line classes
sage: gherkin = dumps(SourPickle([1, 2, 3]))
```

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

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

```python
>>> import __main__
>>> __main__.SourPickle = SourPickle  # a hack to allow us to pickle command-
˓→line classes
>>> gherkin = dumps(SourPickle([Integer(1), Integer(2), Integer(3)]))
```

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

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

```
>>> from sage.all import *
>>> from sage.misc.persist import register_unpickle_override
>>> register_unpickle_override('__main__', 'SourPickle', SweetPickle)
>>> loads(gherkin)
Traceback (most recent call last):
 ...  KeyError: 0
```

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

```
sage: class SweeterPickle(CombinatorialObject, Element):
    ... def __setstate__(self, state):
    ...     # a pickle from CombinatorialObject is just its instance
    ...     # dictionary
    ...     if isinstance(state, dict):
    ...         # this is a fudge: we need an appropriate parent here
    ...         self._set_parent(Tableaux())
    ...         self.__dict__ = state
    ...     else:
    ...         P, D = state
    ...         if P is not None:
    ...             self._set_parent(P)
    ...             self.__dict__ = D
sage: __main__.SweeterPickle = SweeterPickle
sage: register_unpickle_override('__main__', 'SourPickle', SweeterPickle)
```

```
sage: loads(gherkin)  # needs sage.combinat
[1, 2, 3]
sage: loads(dumps(SweeterPickle([1, 2, 3])))  # check that pickles work for...
→SweeterPickle
[1, 2, 3]
```

```
>>> from sage.all import *
>>> class SweeterPickle(CombinatorialObject, Element):
...     def __setstate__(self, state):
...         # a pickle from CombinatorialObject is just its instance
...         # dictionary
...         if isinstance(state, dict):
...             # this is a fudge: we need an appropriate parent here
(... continues on next page)
```

Chapter 5. Saving and Loading Sage Objects

530
The state passed to \_setstate\_() will usually be something like the instance dictionary of the pickled object, however, with some older classes such as CombinatorialObject it will be a tuple. In general, the state can be any python object. Sage provides a special tool, \_explain\_pickle(), which can help in figuring out the contents of an old pickle. Here is a second example.

\begin{verbatim}
 sage: class A():
 ....:   def \_init\_(self, value):
 ....:     self.original_attribute = value
 ....:   def \_repr\_(self):
 ....:     return 'A(%s)' % self.original_attribute
 sage: class B():
 ....:   def \_init\_(self, value):
 ....:     self.new_attribute = value
 ....:   def \_setstate\_(self, state):
 ....:     try:
 ....:       self.new_attribute = state[\'new_attribute\']
 ....:     except KeyError:
 ....:       self.new_attribute = state[\'original_attribute\']
 ....:   def \_repr\_(self):
 ....:     return 'B(%s)' % self.new_attribute
 sage: import \_main__
 sage: \_main__._A = A
 sage: \_main__._B = B
 sage: A(10)
 A(10)
 sage: loads(dumps(A(10)))
 A(10)
 sage: sage.misc.explain_pickle.explain_pickle(dumps(A(10)))
 pg_A = unpickle_global(\'_main__', 'A')
 si = unpickle_newobj(pg_A, ()
 pg_make_integer = unpickle_global('sage.rings.integer', 'make_integer')
 unpickle_build(si, ('original_attribute':pg_make_integer('a'))
 si
 sage: from sage.misc.persist import register_unpickle_override
 sage: register_unpickle_override(\'_main__', 'A', B)
 sage: loads(dumps(\'_main__', 'A'))
 A(10)
 sage: loads(dumps(B(10)))
 B(10)
\end{verbatim}
Utilities, Release 10.4

>>> from sage.all import *
>>> class A():
...     def __init__(self, value):
...         self.original_attribute = value
...     def __repr__(self):
...         return 'A(%s)' % self.original_attribute
>>> class B():
...     def __init__(self, value):
...         self.new_attribute = value
...     def __setstate__(self, state):
...         try:
...             self.new_attribute = state['new_attribute']
...         except KeyError:
...             self.new_attribute = state['original_attribute']
...     def __repr__(self):
...         return 'B(%s)' % self.new_attribute
>>> import __main__
>>> # a hack to allow us to pickle command line classes
>>> __main__.A = A
>>> __main__.B = B
>>> A(Integer(10))
A(10)
>>> loads(dumps(A(Integer(10))))
A(10)
>>> sage.misc.explain_pickle.explain_pickle(dumps(A(Integer(10))))
pg_A = unpickle_global(__main__, A)
si = unpickle_newobj(pg_A, ()
pickle_build(si, {'original_attribute':pg_make_integer(a)})
si
>>> from sage.misc.persist import register_unpickle_override
>>> register_unpickle_override(__main__, A, B)
>>> loads(dumps(A(Integer(10))))
B(10)
>>> loads(dumps(B(Integer(10))))
B(10)

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

```python
def __reduce__(self):
    # Including the documentation properly causes a doc-test failure so we include...
    # it as a comment:'
    # * This is used when pickling integers.
    # * EXAMPLES:
    #   sage: n = 5
    #   sage: t = n.__reduce__(); t
    #   <built-in function make_integer>, (5,)
    #   sage: t[0](*t[1])
    #   5
    #   sage: loads(dumps(n)) == n
    #   True
```

(continues on next page)
This single line below took me HOURS to figure out.
# It is the *trick* needed to pickle Cython extension types.
# The trick is that you must put a pure Python function
# as the first argument, and that function must return
# the result of unpickling with the argument in the second
# tuple as input. All kinds of problems happen
# if we don’t do this.

```python
return sage.rings.integer.make_integer, (self.str(32),)
```

sage.misc.persist.save(obj, filename, compress=True, **kwargs)

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

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

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

### EXAMPLES:

```python
sage: import tempfile
sage: d = tempfile.TemporaryDirectory()
sage: a = matrix(2, [1,2, 3,-5/2])  #...
   → needs sage.modules
sage: objfile = os.path.join(d.name, 'test.sobj')
sage: objfile_short = os.path.join(d.name, 'test')
sage: save(a, objfile)  #...
   → needs sage.modules
sage: load(objfile_short)  #...
   → needs sage.modules
[ 1  2]
[ 3 −5/2]

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

sage: save("A python string", os.path.join(d.name, 'test'))
sage: load(objfile)
'A python string'
sage: load(objfile_short)
'A python string'
sage: d.cleanup()```

5.2. Object persistence
```python
>>> from sage.all import *
>>> import tempfile

>>> d = tempfile.TemporaryDirectory()
>>> a = matrix(Integer(2), [Integer(1), Integer(2), Integer(3), -Integer(5)/Integer(2)])
# needs sage.modules

>>> objfile = os.path.join(d.name, 'test.sobj')
>>> objfile_short = os.path.join(d.name, 'test')

>>> save(a, objfile)  # needs sage.modules

>>> load(objfile_short)  # needs sage.modules

[ 1  2]
[ 3 -5/2]

>>> # needs sage.plot sage.schemes
>>> E = EllipticCurve([-Integer(1), Integer(0)])
>>> P = plot(E)

>>> save(P, objfile_short)  # saves the plot to "test.sobj"

>>> save(P, filename=os.path.join(d.name, "sage.png"), xmin=-Integer(2))

>>> save(P, os.path.join(d.name, "filename.with.some.wrong.ext"))

Traceback (most recent call last):
... ValueError: allowed file extensions for images are
'.eps', '.pdf', '.pgf', '.png', '.ps', '.sobj', '.svg'!

>>> print(load(objfile))
Graphics object consisting of 2 graphics primitives

>>> print(load(objfile_short))
'A python string'

>>> save("A python string", os.path.join(d.name, 'test'))

sage.misc.persist.unpickle_all(target, debug=False, run_test_suite=False)

Unpickle all .sobj files in a directory or tar archive.

INPUT:

- **target** – a string: the name of a directory or of a (possibly compressed) tar archive that contains a single
directory of .sobj files. The tar archive can be in any format that python's tarfile module understands;
for example, .tar.gz or .tar.bz2.

- **debug** – a boolean (default: False) whether to report a stacktrace in case of failure

- **run_test_suite** – a boolean (default: False) whether to run TestSuite(x).run() on the un-
pickled objects

OUTPUT:

Typically, two lines are printed: the first reporting the number of successfully unpickled files, and the second
reporting the number (zero) of failures. If there are failures, however, then a list of failed files will be printed
before either of those lines, and the failure count will of course be non-zero.

**Warning:** You must only pass trusted data to this function, including tar archives. We use the “data” filter
from PEP 706 if possible while extracting the archive, but even that is not a perfect solution, and it is only
available since Python 3.11.4.
```
EXAMPLES:

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

```
>>> from sage.all import *
>>> dir = tmp_dir()
>>> sage.misc.persist.picklejar('hello', dir)
>>> sage.misc.persist.unpickle_all(dir)
Successfully unpickled 1 objects.
Failed to unpickle 0 objects.
```

`sage.misc.persist.unpickle_global(module, name)`

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

EXAMPLES:

```
sage: from sage.misc.persist import unpickle_override, register_unpickle_override
sage: unpickle_global('sage.rings.integer', 'Integer')
<class 'sage.rings.integer.Integer'>
```

```
>>> from sage.all import *
>>> from sage.misc.persist import unpickle_override, register_unpickle_override
>>> unpickle_global('sage.rings.integer', 'Integer')
<class 'sage.rings.integer.Integer'>
```

Now we horribly break the pickling system:

```
sage: register_unpickle_override('sage.rings.integer', 'Integer', Rational, call_-˓→name=('sage.rings.rational', 'Rational'))
sage: unpickle_global('sage.rings.integer', 'Integer')
<class 'sage.rings.rational.Rational'>
```

```
>>> from sage.all import *
>>> register_unpickle_override('sage.rings.integer', 'Integer', Rational, call_-˓→name=('sage.rings.rational', 'Rational'))
>>> unpickle_global('sage.rings.integer', 'Integer')
<class 'sage.rings.rational.Rational'>
```

and we reach into the internals and put it back:

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

```
>>> from sage.all import *
>>> del unpickle_override[('sage.rings.integer', 'Integer')]
>>> unpickle_global('sage.rings.integer', 'Integer')
<class 'sage.rings.integer.Integer'>
```
A meaningful error message with resolution instructions is displayed for old pickles that accidentally got broken because a class or entire module was moved or renamed:

```python
sage: unpickle_global('sage.all', 'some_old_class')
Traceback (most recent call last):
  ...  
ImportError: cannot import some_old_class from sage.all, call
register_unpickle_override('sage.all', 'some_old_class', ...)
to fix this

sage: unpickle_global('sage.some_old_module', 'some_old_class')
Traceback (most recent call last):
  ...  
ImportError: cannot import some_old_class from sage.some_old_module, call
register_unpickle_override('sage.some_old_module', 'some_old_class', ...)
to fix this
```

```python
def from sage.all import *
def foo(a,b,c=10): return a+b+c
```

```python
def foo(a,b,c=Integer(10)): return a+b+c
```

5.3 Function pickling

REFERENCE: The python cookbook.

sage.misc.fpickle.call_pickled_function(fpargs)

sage.misc.fpickle.code_ctor(*args)

EXAMPLES:

This indirectly tests this function.

```python
sage: def foo(a,b,c=10): return a+b+c
sage: sage.misc.fpickle.reduce_code(foo.__code__)
<cyfunction code_ctor at 0x3e8ba5f9, ...>
sage: unpickle_function(pickle_function(foo))
<function foo at 0x3e8b21f9>
```

```python
>>> from sage.all import *
>>> def foo(a,b,c=Integer(10)): return a+b+c
>>> sage.misc.fpickle.reduce_code(foo.__code__)
<cyfunction code_ctor at 0x3e8ba5f9, ...>
>>> unpickle_function(pickle_function(foo))
<function foo at 0x3e8b21f9>
```
sage.misc.fpickle.pickleMethod(method)
support function for copyreg to pickle method refs

sage.misc.fpickle.pickleModule(module)
support function for copyreg to pickle module refs

sage.misc.fpickle.pickle_function(func)

Pickle the Python function func. This is not a normal pickle; you must use the unpickle_function method to unpickle the pickled function.

NOTE: This does not work on all functions, but does work on 'surprisingly' many functions. In particular, it does not work on functions that includes nested functions.

INPUT:
  func -- a Python function

OUTPUT:
  a string

EXAMPLES:

```python
sage: def f(N):
    return N+1
...
sage: g = pickle_function(f)
sage: h = unpickle_function(g)
sage: h(10)
11
```

```python
>>> from sage.all import *
>>> def f(N):
    return N+Integer(1)
...
>>> g = pickle_function(f)
>>> h = unpickle_function(g)
>>> h(Integer(10))
11
```

sage.misc.fpickle.reduce_code(co)

EXAMPLES:

```python
sage: def foo(N):
    return N+1
sage: reduce_code = reduce_code(foo.__code__)
(<cyfunction code_ctor at ...>, ...)
```

```python
>>> from sage.all import *
>>> def foo(N):
    return N+Integer(1)

>>> reduce_code = reduce_code(foo.__code__)
(<cyfunction code_ctor at ...>, ...)
```

Test that the constructed code matches the original code:

```python
sage: ctor, args = reduce_code
sage: ctor(*args) == foo
True
```

```python
>>> from sage.all import *
>>> ctor, args = reduce_code
```

(continues on next page)
sage.misc.fpickle.unpickleMethod\((im\_name, \_self, im\_class)\)
support function for copyreg to unpickle method refs

sage.misc.fpickle.unpickleModule\((name)\)
support function for copyreg to unpickle module refs

sage.misc.fpickle.unpickle_function\((pickled)\)
Unpickle a pickled function.

**EXAMPLES:**

```python
sage: def f(N,M): return N*M
...
```

```python
sage: unpickle_function(pickle_function(f))(3,5)
15
```

```python
>>> from sage.all import *
>>> def f(N,M): return N*M
...
```

```python
>>> unpickle_function(pickle_function(f))(Integer(3),Integer(5))
15
```

5.4 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:**

```python
sage: explain_pickle(dumps(12345))
gp_make_integer = unpickle_global('sage.rings.integer', 'make_integer')
gp_make_integer('c1p')
sage: explain_pickle(dumps(polygen(QQ)))
pg_Polynomial_rational_flint = unpickle_global('sage.rings.polynomial.polynomial_rational_flint', 'Polynomial_rational_flint')
polynomials = pg_unpickle_PolynomialRing(pg_RationalField, (x,), None, False, True)
pg = unpickle_instantiate(pg_RationalField, ())
gp_make_rational = unpickle_global('sage.rings.rational', 'make_rational')
pg_Polynomial_rational_flint(pg_unpickle_PolynomialRing(pg, ('x'),), None, False), [pg_ ˓→make_rational('0'), pg_make_rational('1')], False, True)
sage: sage_eval(explain_pickle(dumps(polygen(QQ)))) == polygen(QQ)
True
```
Utilities, Release 10.4

>>> from sage.all import *
>>> explain_pickle(dumps(Integer(12345)))
pg_make_integer = unpickle_global('sage.rings.integer', 'make_integer')
pg_make_integer('c1p')
>>> explain_pickle(dumps(polygen(QQ)))
pg_Polynomial_rational_flint = unpickle_global('sage.rings.polynomial.polynomial_rational_flint', 'Polynomial_rational_flint')
pg_unpickle_PolynomialRing = unpickle_global('sage.rings.polynomial.polynomial_ring_constructor', 'unpickle_PolynomialRing')
pg_RationalField = unpickle_global('sage.rings.rational_field', 'RationalField')
pg = unpickle_instantiate(pg_RationalField, ())
pg_make_rational = unpickle_global('sage.rings.rational', make_rational)
polynomial_rational_flint(pg_unpickle_PolynomialRing(pg, (x,), None, False), [pg_make_rational(0), pg_make_rational(1)], False, True)
>>> 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:

```python
sage: explain_pickle(dumps(12345), in_current_sage=True)
from sage.rings.integer import make_integer
make_integer('c1p')
```

```python
sage: explain_pickle(dumps(polygen(QQ)), in_current_sage=True)
from sage.rings.polynomial.polynomial_rational_flint import Polynomial_rational_flint
from sage.rings.polynomial.polynomial_ring_constructor import unpickle_PolynomialRing
from sage.rings.rational import make_rational
Polynomial_rational_flint(unpickle_PolynomialRing(RationalField(), (x,), None, False), [make_rational(0), make_rational(1)], False, True)
```

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**

5.4. A tool for inspecting Python pickles
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 `explain_pickle` doctest to use the new output), to ensure that old pickles will continue to work.

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.misc.persist.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:

```python
from sage.rings.integer import integer
```

and if you ran `explain_pickle()` in hypothetical future sage, it would generate:

```python
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:

```python
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
    Bases: object
    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 SageInputExpression objects. 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 SageInputExpression objects instead of directly constructing values.
```

APPEND()
APPENDS()
BINFLOAT(f)
BINGET(n)
BININT(n)
BININT1(n)
BININT2(n)
BINPERSID()
BININPUT(n)
BINSTRING(s)
BINUNICODE(s)
BUILD()
DICT()
DUP()
EMPTY_DICT()
EMPTY_LIST()
EMPTY_TUPLE()
EXT1(n)
EXT2(n)
EXT4(n)
FLOAT(f)
GET(n)
GLOBAL(name)
INST(name)
INT(n)
LIST()
LONG(n)
LONG1(n)
LONG4(n)
LONG_BINGET(n)
LONG_BINPUT(n)
MARK()
NEWFALSE()
NEWOBJ()
NEWTRUE()
NONE()
OBJ()
PERSID(id)
POP()
POP_MARK()
PROTO(proto)
PUT(n)
REDUCE()
SETITEM()
SETITEMS()
SHORT_BINSTRING(s)
STOP()
STRING(s)
Utilities, Release 10.4

TUPLE()

TUPLE1()

TUPLE2()

TUPLE3()

UNICODE(s)

check_value(v)

Check that the given value is either a SageInputExpression or a PickleObject. Used for internal sanity checking.

EXAMPLES:

```python
from sage.misc.explain_pickle import *
from sage.misc.sage_input import SageInputBuilder
sib = SageInputBuilder()
pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=True)
pe.check_value(7)
Traceback (most recent call last):
...ASSERTIONERROR
sage: pe.check_value(sib(7))
```

```python
>>> from sage.all import *
>>> from sage.misc.explain_pickle import *
>>> from sage.misc.sage_input import SageInputBuilder
>>> sib = SageInputBuilder()
>>> pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=True)
>>> pe.check_value(Integer(7))
Traceback (most recent call last):
...ASSERTIONERROR
>>> pe.check_value(sib(Integer(7)))
```

is_mutable_pickle_object(v)

Test whether a PickleObject is mutable (has never been converted to a SageInputExpression).

EXAMPLES:

```python
from sage.misc.explain_pickle import *
from sage.misc.sage_input import SageInputBuilder
sib = SageInputBuilder()
pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=True)
v = PickleObject(1, sib(!))
pe.is_mutable_pickle_object(v)
True
sage: sib(v)
{\text{atomic:1}}
sage: pe.is_mutable_pickle_object(v)
False
```
pop()

Pop a value from the virtual machine’s stack, and return it.

EXAMPLES:

```python
>>> from sage.all import *
>>> from sage.misc.explain_pickle import *
>>> from sage.misc.sage_input import SageInputBuilder
>>> sib = SageInputBuilder()
>>> pe = PickleExplainer(sib, in_current_sage=True,
                        ... default_assumptions=False, pedantic=True)
>>> v = PickleObject(Integer(1), sib(Integer(1)))
>>> pe.is_mutable_pickle_object(v)
True
>>> sib(v)
{atomic:1}
>>> pe.is_mutable_pickle_object(v)
False
```
push (v)

Push a value onto the virtual machine’s stack.

EXAMPLES:

```python
>>> from sage.misc.explain_pickle import *
>>> from sage.misc.sage_input import SageInputBuilder
>>> sib = SageInputBuilder()
>>> pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False,
  pedantic=True)
>>> pe.push(sib(Integer(7)))
>>> 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:

```python
>>> from sage.all import *
>>> from sage.misc.explain_pickle import *
>>> from sage.misc.sage_input import SageInputBuilder
>>> sib = SageInputBuilder()
>>> pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False,
  pedantic=True)
>>> pe.push_and_share(sib(7))
>>> pe.stack[-1]
{atomic:7}
```
push_mark()

Push a ‘mark’ onto the virtual machine’s stack.

EXAMPLES:

```python
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=True)
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:

```python
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=True)
sage: sib(pe.run_pickle('T\5\0\0\0hello.'))  # py2
{atomic:'hello'}
```

```python
sage: from sage.all import *
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=True)
```
share(v)

Mark a sage_input() value as shared, if we are in pedantic mode.

EXAMPLES:

```python
>>> sib(pe.run_pickle('T\5\0\0\0\0hello.'))  # py2
{atomic:'hello'}
```

```python
>>> from sage.misc.explain_pickle import *
>>> from sage.misc.sage_input import *
>>> from sage.misc.explain_pickle import *
>>> from sage.misc.sage_input import *
>>> sib = SageInputBuilder()
>>> pe = PickleExplainer(sib, in_current_sage=True,
...     default_assumptions=False, pedantic=True)
>>> v = sib(7)
>>> v._sie_share
False
>>> pe.share(v)
{atomic:7}
>>> v._sie_share
True
```

```python
>>> from sage.all import *
>>> from sage.misc.explain_pickle import *
>>> from sage.misc.sage_input import *
>>> sib = SageInputBuilder()
>>> pe = PickleExplainer(sib, in_current_sage=True,
...     default_assumptions=False, pedantic=True)
>>> v = sib(Integer(7))
>>> v._sie_share
False
>>> pe.share(v)
{atomic:7}
>>> 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 sage.misc.sage_input.SageInputExpression objects 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(iterable=(), /)

Bases: list
A subclass of \texttt{list}, with deliberately-broken append and extend methods. Used for testing \texttt{explain_pickle()}. 

\texttt{append()} 

A deliberately broken append method. 

\textbf{EXAMPLES:} 

\begin{verbatim}
sage: from sage.misc.explain_pickle import *
sage: v = TestAppendList()
sage: v.append(7)
Traceback (most recent call last):
...
TypeError: ...append() takes 1 positional argument but 2 were given

>>> from sage.all import *
>>> from sage.misc.explain_pickle import *
>>> v = TestAppendList()
>>> v.append(Integer(7))
Traceback (most recent call last):
...
TypeError: ...append() takes 1 positional argument but 2 were given

We can still append by directly using the list method:

\begin{verbatim}
sage: list.append(v, 7)
sage: v
[7]
\end{verbatim}

\texttt{extend()} 

A deliberately broken extend method. 

\textbf{EXAMPLES:} 

\begin{verbatim}
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 1 positional argument but 2 were given

>>> from sage.all import *
>>> from sage.misc.explain_pickle import *
>>> v = TestAppendList()
>>> v.extend([Integer(3),Integer(1),Integer(4),Integer(1),Integer(5),
           Integer(9)])
Traceback (most recent call last):
...
TypeError: ...extend() takes 1 positional argument but 2 were given

We can still extend by directly using the list method:

\end{verbatim}
sage: list.extend(v, (3,1,4,1,5,9))
sage: v
[3, 1, 4, 1, 5, 9]

>>> from sage.all import *
>>> list.extend(v, (Integer(3),Integer(1),Integer(4),Integer(1),Integer(5),
   →Integer(9)))
>>> 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: 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

>>> from sage.all import *
>>> from sage.misc.explain_pickle import *
>>> globals()['funny$name'] = TestGlobalFunnyName # see comment at end of file
>>> TestGlobalFunnyName.__name__
'funny$name'
>>> 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

5.4. A tool for inspecting Python pickles 549
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 \texttt{TestGlobalNewName} instead. Used for testing \texttt{explain_pickle()}.

EXAMPLES:

```python
going from sage.misc.explain_pickle import *
loads(dumps(TestGlobalOldName()))
TestGlobalNewName
```

class sage.misc.explain_pickle.TestReduceGetinitargs

Bases: object

An old-style class with a \texttt{\_getinitargs\_()} method. Used for testing \texttt{explain_pickle()}.

class sage.misc.explain_pickle.TestReduceNoGetinitargs

Bases: object

An old-style class with no \texttt{\_getinitargs\_()} method. Used for testing \texttt{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 \texttt{explain_pickle()} to \texttt{sage_eval()} should be totally equivalent to loading the pickle with \texttt{cPickle}.

INPUT:

- \texttt{pickle} -- the pickle to explain, as a string (default: None)
- \texttt{file} -- a filename of a pickle (default: None)
- \texttt{compress} -- if False, don’t attempt to decompress the pickle (default: True)
- \texttt{in\_current\_sage} -- if True, produce potentially simpler code that is tied to the current version of Sage. (default: False)
- \texttt{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)
- \texttt{eval} -- if True, then evaluate the resulting code and return the evaluated result. (default: False)
- \texttt{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)
- \texttt{pedantic} -- if True, then carefully ensures that the result has at least as much sharing as the result of \texttt{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:

```python
sage: explain_pickle(dumps({('a', 'b'): [1r, 2r]}))
{(a', 'b'): [1r, 2r]})

sage: explain_pickle(dumps(RR(pi)), in_current_sage=True)  # needs sage.symbolic
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'),

.sage: explain_pickle(dumps((s, s)))
('hi', 'hi')
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')

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')

>>> from sage.all import *

>>> explain_pickle(dumps({('a', 'b'): [i, 2r]}))
{(a', 'b'): [1r, 2r]})

>>> explain_pickle(dumps(RR(pi)), in_current_sage=True)  # needs sage.symbolic
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'),

>>> s = 'hi'

>>> explain_pickle(dumps((s, s)))
('hi', 'hi')
si = 'hi'
(si, si)

>>> explain_pickle(dumps(5))
5r

>>> explain_pickle(dumps(5), preparse=False)
5

>>> explain_pickle(dumps(5), preparse=None)
int(5)

>>> explain_pickle(dumps(Integer(22)/Integer(7)))
pg_make_rational = unpickle_global('sage.rings.rational', 'make_rational')
```
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: from sage.misc.explain_pickle import explain_pickle_string
sage: explain_pickle_string(dumps("Hello, world",
←compress=False))
'Hello, world'

>>> from sage.all import *
>>> 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

>>> from sage.all import *
>>> from sage.misc.explain_pickle import name_is_valid
>>> name_is_valid('fred')
True
>>> name_is_valid('Yes!ValidName')
False
>>> name_is_valid('_happy_1234')
True

sage.misc.explain_pickle.test_pickle(p, verbose_eval=False, pedantic=False, args=())

Test 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; \texttt{test\_pickle()} will pickle the object

Once it has a pickle, \texttt{test\_pickle()} will print the pickle’s disassembly, run \texttt{explain\_pickle()} with \texttt{in\_current\_sage=True} and \texttt{False}, print the results, evaluate the results, unpickle the object with \texttt{cPickle}, and compare all three results.

If \texttt{verbose\_eval} is \texttt{True}, then \texttt{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:

\begin{verbatim}sage: from sage.misc.explain_pickle import * sage: test_pickle(['a']) 0: \x80 PROTO 2 2: ] EMPTY_LIST 3: q BINPUT 0 5: X BINUNICODE 'a' 11: q BINPUT 1 13: a APPEND 14: . STOP

highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False: ['a']
result: ['a']
\end{verbatim}

\begin{verbatim}>>> from sage.all import *
>>> from sage.misc.explain_pickle import *
>>> test_pickle(['a']) 0: \x80 PROTO 2 2: ] EMPTY_LIST 3: q BINPUT 0 5: X BINUNICODE 'a' 11: q BINPUT 1 13: a APPEND 14: . STOP

highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False: ['a']
result: ['a']
\end{verbatim}

\begin{verbatim}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 \texttt{cPickle} would use. Used by \texttt{explain\_pickle()}.

EXAMPLES:

\begin{verbatim}sage: v = [] sage: unpickle_appends(v, (1, 2, 3)) sage: v
[1, 2, 3]
\end{verbatim}

\begin{verbatim}>>> from sage.all import *
>>> v = []
>>> unpickle_appends(v, (Integer(1), Integer(2), Integer(3)))
>>> v
[1, 2, 3]
\end{verbatim}
Utilities, Release 10.4

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 copyreg 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:
5.5 Fixing pickle for nested classes

As of Python 2.7, names for nested classes are set by Python in a way which is incompatible with the pickling of such classes (pickling by name):

```python
sage: class A:
    ....:     class B:
    ....:         pass
sage: A.B.__name__
'B'
```

Instead of more natural "A.B". Furthermore upon pickling and unpickling 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: https://groups.google.com/forum/#!topic/sage-devel/bHBV9KWAt64

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.test_nested_class`. 
Note: In Python 3, nested classes, like any class for that matter, have \_\_\_qualname\_\_ and the standard pickle module uses it for pickling and unpickling. Thus the pickle module searches for "A.B" first by looking up "A" in mod, and then "B" in the result. So there is no pickling problem for nested classes in Python 3, and the two utilities are not really necessary. However, \_\_\_NestedClassMetaclass\_\_ is used widely in Sage and affects behaviors of Sage objects in other respects than in pickling and unpickling. Hence we keep \_\_\_NestedClassMetaclass\_\_ even with Python 3, for now. This module will be removed when we eventually drop support for Python 2.

EXAMPLES:

```python
sage: from sage.misc.nested_class import A1, nested_pickle

sage: A1.A2.A3.__name__
'A3'
<class 'sage.misc.nested_class.A1.A2.A3'>

sage: nested_pickle(A1)
<class 'sage.misc.nested_class.A1'>

sage: A1.A2
<class 'sage.misc.nested_class.A1.A2'>

<class 'sage.misc.nested_class.A1.A2.A3'>
sage: A1.A2.A3.__name__
'A1.A2.A3'

True
True

>>> from sage.all import *
>>> from sage.misc.nested_class import A1, nested_pickle

>>> A1.A2.A3.__name__
'A3'
<class 'sage.misc.nested_class.A1.A2.A3'>

>>> nested_pickle(A1)
<class 'sage.misc.nested_class.A1'>

>>> A1.A2
<class 'sage.misc.nested_class.A1.A2'>

<class 'sage.misc.nested_class.A1.A2.A3'>
>>> A1.A2.A3.__name__
'A1.A2.A3'

>>> sage.misc.nested_class.__dict__['A1.A2'] is A1.A2
True
True
```
All of this is not perfect. In the following scenario:

```python
sage: class A1:
    ....:    class A2:
    ....:        pass
sage: class B1:
    ....:        A2 = A1.A2

sage: nested_pickle(A1)
<class '__main__.A1'>
sage: nested_pickle(B1)
<class '__main__.B1'>
sage: A1.A2
<class '__main__.A1.A2'>
sage: B1.A2
<class '__main__.A1.A2'>
```

```python
>>> from sage.all import *
>>> class A1:
...    class A2:
...        pass
>>> class B1:
...        A2 = A1.A2
```

```python
>>> nested_pickle(A1)
<class '__main__.A1'>
>>> nested_pickle(B1)
<class '__main__.B1'>
>>> A1.A2
<class '__main__.A1.A2'>
>>> B1.A2
<class '__main__.A1.A2'>
```

The name for "A1.A2" could potentially be set to "B1.A2". But that will work anyway.

```python
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...>
```

```python
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...>
```

5.5. Fixing pickle for nested classes
Utilities, Release 10.4

```python
>>> from sage.all import *
>>> from sage.misc.nested_class import *
>>> loads(dumps(MainClass.NestedClass()))
<sage.misc.nested_class.MainClass.NestedClass object at 0x...>
```

```python
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'
    >>> from sage.all import *
    >>> from sage.misc.nested_class import *
    >>> loads(dumps(MainClass.NestedClass.NestedSubClass()))
    <sage.misc.nested_class.MainClass.NestedClass.NestedSubClass object at 0x...>
    >>> getattr(sage.misc.nested_class, MainClass.NestedClass.NestedSubClass)
    <class sage.misc.nested_class.MainClass.NestedClass.NestedSubClass>
    >>> MainClass.NestedClass.NestedSubClass.__name__
    'MainClass.NestedClass.NestedSubClass'
```

dummy ($x, r=(1, 2, 3.4), *args, **kwds$)

A dummy method to demonstrate the embedding of method signature for nested classes.

```python
class sage.misc.nested_class.NestedClassMetacllass
    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 NestedClassMetacllass
    sage: class ASuperClass(object, metaclass=NestedClassMetacllass):
    ....:    pass
    sage: class A3(ASuperClass):
    ....:        class B():
    ....:            pass
    sage: A3.B.__name__
    'A3.B'
    sage: getattr(sys.modules['__main__'], 'A3.B', 'Not found')
    <class '__main__.A3.B'>
    >>> from sage.all import *
    >>> from sage.misc.nested_class import NestedClassMetacllass
    >>> class ASuperClass(object, metaclass=NestedClassMetacllass):
    ...    pass
    >>> class A3(ASuperClass):
    ...    (continues on next page)
```
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.

INPUT:

• 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:

```python
>>> from sage.misc.nested_class import *
>>> class A():
...    class B():
...        pass
>>> module = sys.modules['__main__']
>>> A.B.__name__
'B'
>>> getattr(module, 'A.B', 'Not found')
'Not found'
>>> modify_for_nested_pickle(A, 'A', module)
>>> A.B.__name__
'A.B'
>>> getattr(module, 'A.B', 'Not found')
<class '__main__.A.B'>
```

Here we demonstrate the effect of the first_run argument:

```python
>>> from sage.all import *
>>> from sage.misc.nested_class import *
>>> class A():
...    class B():
...        pass
>>> module = sys.modules['__main__']
>>> A.B.__name__
'B'
>>> getattr(module, 'A.B', 'Not found')
'Not found'
>>> modify_for_nested_pickle(A, 'A', module)
>>> A.B.__name__
'A.B'
>>> getattr(module, 'A.B', 'Not found')
<class '__main__.A.B'>
```
sage: modify_for_nested_pickle(A, 'X', module)
sage: A.B.__name__  # nothing changed
'A.B'
sage: modify_for_nested_pickle(A, 'X', module, first_run=False)
sage: A.B.__name__
'X.A.B'

>>> from sage.all import *
>>> modify_for_nested_pickle(A, 'X', module)
>>> A.B.__name__  # nothing changed
'A.B'
>>> modify_for_nested_pickle(A, 'X', module, first_run=False)
>>> 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__.A.B>
sage: getattr(module, X.A.B, Not found)
<class __main__.A.B>

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:

```python
sage: from sage.misc.nested_class import nested_pickle
sage: module = sys.modules['__main__']
sage: class A():
    ....:    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":

```python
sage: A.B.__name__
'A.B'
sage: getattr(module, A.B, 'Not found')
<class '__main__.A.B'>
```
>>> from sage.all import *
>>> A.B.__name__
'A.B'
>>> getattr(module, 'A.B', 'Not found')
<class '__main__.A.B'>

In Python 2.6, decorators work with classes; then `@nested_pickle` should work as a decorator:

```python
sage: @nested_pickle
....: class A2():
....:     class B:
....:         pass
sage: A2.B.__name__
'A2.B'
>>> getattr(module, 'A2.B', 'Not found')
<class __main__.A2.B at ...>
```

```python
>>> from sage.all import *
>>> @nested_pickle
... class A2():
...     class B:
...         pass
>>> A2.B.__name__
# todo: not implemented
A2.B
```

EXAMPLES:

```python
sage: from sage.misc.nested_class import *
sage: loads(dumps(MainClass.NestedClass()))
# indirect doctest
<sage.misc.nested_class.MainClass.NestedClass object at 0x...>
```

```python
>>> from sage.all import *
>>> from sage.misc.nested_class import *
>>> loads(dumps(MainClass.NestedClass()))
# indirect doctest
<sage.misc.nested_class.MainClass.NestedClass object at 0x...>
```

5.6. 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:

```python
sage: reset()
sage: w = 2/3; w
2/3
sage: show_identifiers()
['w']
```

```python
>>> from sage.all import *
>>> reset()
>>> w = Integer(2)/Integer(3); w
2/3
(continues on next page)
```
We next save this session. We are using a temporary directory to hold the session file but we do this for testing only. Please do not do this if you want to save your session permanently. Also note that the `tempfile` module weasels its way into the session:

```python
sage: from tempfile import TemporaryDirectory
dsage: d = TemporaryDirectory()
sage: save_session(os.path.join(d.name, 'session'))
```

This saves a dictionary with `w` as one of the keys:

```python
sage: z = load(os.path.join(d.name, 'session'))
sage: list(z)
[w, d]
sage: z['w']
2/3
```

Next we reset all variables in the session except for the temporary directory name. We verify that the session is reset, and then load it back:

```python
sage: sage.misc.reset.EXCLUDE.add(d)
sage: reset()
sage: show_identifiers()
['d']
sage: load_session(os.path.join(d.name, 'session'))
```

Indeed `w` is now defined again:

```python
sage: show_identifiers()
['d', 'w']
sage: w
2/3
```
Finally, we clean up the temporary directory:

```python
sage: d.cleanup()
```

```python
>>> from sage.all import *

```  

```python
>>>

```  

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:

```python
sage: reset()

```  

```python
sage: w = 10

```  

```python
sage: show_identifiers()

w

```  

```python
[]

```  

When we call `init()` below it reinitializes the internal table, so the `w` we just defined doesn’t count as a new identifier:

```python
sage: sage.misc.session.init()

```  

```python
sage: show_identifiers()

[]

```  

```python
[]

```  

`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:
Utilities, Release 10.4

```python
sage: a = 5
sage: f = lambda x: x^2

>>> from sage.all import *
>>> a = Integer(5)
>>> f = lambda x: x**Integer(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.

```python
sage: tmp_f = tmp_filename()
sage: save_session(tmp_f)
sage: del a; del f
sage: load_session(tmp_f)
sage: print(a)
5

```python
>>> from sage.all import *
>>> tmp_f = tmp_filename()
>>> save_session(tmp_f)
>>> del a; del f
>>> load_session(tmp_f)
>>> print(a)
5

Note that \( f \) does not come back, since it is a function, hence couldn’t be saved:

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

>>> from sage.all import *
>>> 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. One can still make sessions that can’t be reloaded. E.g., define a class with:

   ```python
class Foo: pass
   ```

   and make an instance with:

   ```python
   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
```

```python
>>> from sage.all import *
```  
```python
>>> a = Integer(5)
```  
```python
>>> tmp_f = tmp_filename()
```  
```python
>>> save_session(tmp_f)
```  
```python
>>> del a
```  
```python
>>> load_session(tmp_f)
```  
```python
>>> 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, verbose=True)
...
Not saving f: f is a function or method
```

```python
>>> from sage.all import *
```  
```python
>>> f = lambda x : x**Integer(2)
```  
```python
>>> save_session(tmp_f, verbose=True)
```  
```python
...
Not saving f: f is a function or method
```

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 cython function or method
```
Utilities, Release 10.4

```python
>>> from sage.all import *
>>> g = cython_lambda('double x', 'x*x + 1.5')
>>> save_session(tmp_f, verbose=True)
...
Not saving g: g is a cython function or method
```

And the same for a lazy import:

```python
sage: from sage.misc.lazy_import import LazyImport
sage: lazy_ZZ = LazyImport('sage.rings.integer_ring', 'ZZ')
sage: save_session(tmp_f, verbose=True)
...
Not saving lazy_ZZ: lazy_ZZ is a lazy import
```

```python
>>> from sage.all import *
>>> from sage.misc.lazy_import import LazyImport
>>>
lazy_ZZ = LazyImport('sage.rings.integer_ring', 'ZZ')
>>> save_session(tmp_f, verbose=True)
...
Not saving lazy_ZZ: lazy_ZZ is a lazy import
```

`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:

```python
sage: reset()
sage: show_identifiers()
[]
```

```python
>>> from sage.all import *
>>> reset()
>>> show_identifiers()
[]
```

We then define two variables, one which overwrites the default factor function; both are shown by `show_identifiers()`:

```python
sage: a = 10
sage: factor = 20
sage: show_identifiers()
['factor', 'a']
```

```python
>>> from sage.all import *
>>> a = Integer(10)
>>> factor = Integer(20)
```

(continues on next page)
To get the actual value of a variable from the list, use the `globals()` function:

```python
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:

```python
sage: _hello = 10
sage: show_identifiers()
['factor', 'a']
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:

```python
sage: show_identifiers(hidden=True)     # random output
['__builtin__', '_ih', '_oh', '_dh', 'exit', 'quit', '_', '__', '___', '_i', '_ii', '_iii', '_i1', 'factor', '_i2', '_2', '_i3', 'a', '_i4', '_i5', '_5', '_i6', '_6', '_i7', '_hello', '_i8', '_8', '_i9', '_9', '_i10']
```

```python
>>> from sage.all import *
>>> show_identifiers(hidden=True)     # random output
['__builtin__', '_ih', '__', '___', '_i', '_ii', '_iii', '_i1', 'factor', '_i2', '_2', '_i3', 'a', '_i4', '_i5', '_5', '_i6', '_6', '_i7', '_hello', '_i8', '_8', '_i9', '_9', '_i10']
```
6.1 Interactive Sage Sessions

6.1.1 SageMath version and banner info

sage.misc.banner.banner()

Print the Sage banner.

OUTPUT: None

If the environment variable SAGE_BANNER is set to no, no banner is displayed. If SAGE_BANNER is set to bare, a simplified plain ASCII banner is displayed. Otherwise, the full banner with box art is displayed.

EXAMPLES:

```
sage: import sage.misc.banner; sage.misc.banner.SAGE_BANNER = ''
sage: sage.misc.banner.banner()
```

```
┌────────────────────────────────────────────────────────────────────┐
│ SageMath version ..., Release Date: ... │
│ Using Python .... Type "help()" for help. │
...```

```
>>> from sage.all import *
>>> import sage.misc.banner; sage.misc.banner.SAGE_BANNER = ''
>>> sage.misc.banner.banner()
```

```
┌────────────────────────────────────────────────────────────────────┐
│ SageMath version ..., Release Date: ... │
│ Using Python .... Type "help()" for help. │
...```

sage.misc.banner.banner_text(full=True)

Text for the Sage banner.

INPUT:

  • full – boolean (default: True)

OUTPUT:

A string containing the banner message.

If option full is False, a simplified plain ASCII banner is displayed; if True the full banner with box art is displayed.

EXAMPLES:
sage: print(sage.misc.banner.banner_text(full=True))
| SageMath version ...

sage: print(sage.misc.banner.banner_text(full=False))
SageMath version ..., Release Date: ...

>>> from sage.all import *
>>> print(sage.misc.banner.banner_text(full=True))
| SageMath version ...

>>> print(sage.misc.banner.banner_text(full=False))
SageMath version ..., Release Date: ...

sage.misc.banner.require_version(major, minor=0, tiny=0, prerelease=False, print_message=False)

True if Sage version is at least major.minor.tiny.

INPUT:

- major – integer
- minor – integer (default: 0)
- tiny – float (default: 0)
- prerelease – boolean (default: False)
- print_message – boolean (default: False)

OUTPUT:

True if major.minor.tiny is <= version of Sage, False otherwise

For example, if the Sage version number is 3.1.2, then require_version(3, 1, 3) will return False, while require_version(3, 1, 2) will return True. If the Sage version is 3.1.2.alpha0, then require_version(3, 1, 1) will return True, while, by default, require_version(3, 1, 2) will return False. Note, though, that require_version(3, 1, 2, prerelease=True) will return True: if the optional argument prerelease is True, then a prerelease version of Sage counts as if it were the released version.

If optional argument print_message is True and this function is returning False, print a warning message.

EXAMPLES:

sage: from sage.misc.banner import require_version
sage: require_version(2, 1, 3)
True
sage: require_version(821, 4)
False
sage: require_version(821, 4, print_message=True)
This code requires at least version 821.4 of SageMath to run correctly.
You are running version ...
False

(continues on next page)
You are running version ...
False

sage.misc.banner.version()

Return the version of Sage.

OUTPUT:
str

EXAMPLES:

```python
sage: version()
'SageMath version ..., Release Date: ...
```

sage.misc.banner.version_dict()

A dictionary describing the version of Sage.

INPUT:
nothing

OUTPUT:

This process the Sage version string and produces a dictionary. It expects the Sage version to be in one of these forms:

- N.N
- N.N.N
- N.N.N.N
- N.N.str
- N.N.N.str
- N.N.N.N.str

where ‘N’ stands for an integer and ‘str’ stands for a string. The first integer is stored under the ‘major’ key and the second integer under ‘minor’. If there is one more integer, it is stored under ‘tiny’; if there are two more integers, then they are stored together as a float N.N under ‘tiny’. If there is a string, then the key ‘prerelease’ returns True.

For example, if the Sage version is ‘3.2.1’, then the dictionary is {'major': 3, 'minor': 2, 'tiny': 1, 'prerelease': False}. If the Sage version is ‘3.2.1.2’, then the dictionary is {'major': 3, 'minor': 2, 'tiny': 1.200..., 'prerelease': False}. If the Sage version is ‘3.2.alpha0’, then the dictionary is {'major': 3, 'minor': 2, 'tiny': 0, 'prerelease': True}.

EXAMPLES:

```python
sage: from sage.misc.banner import version_dict
sage: version_dict()['major'] == int(sage.version.version.split('.')[0])
```

```python
>>> from sage.all import *
>>> version_dict()
```
6.1.2 Interpreter reset

`sage.misc.reset.reset`(*vars=None, attached=False*)

Delete all user-defined variables, reset all global variables back to their default states, and reset all interfaces to other computer algebra systems.

If `vars` is specified, just restore the value of `vars` and leave all other variables alone (i.e., call `restore`).

Note that the variables in the set `sage.misc.reset.EXCLUDE` are excluded from being reset.

**INPUT:**

- `vars` — a list, or space or comma separated string (default: `None`), variables to restore
- `attached` — boolean (default: `False`), if `vars` is not `None`, whether to detach all attached files

**EXAMPLES:**

```plaintext
sage: x = 5
sage: reset()

# needs sage.symbolic
x

sage: fn = tmp_filename(ext='.py')
sage: sage.misc.reset.EXCLUDE.add('fn')
sage: with open(fn, 'w') as f:
    _ = f.write('a = 111')
sage: attach(fn)
sage: af = attached_files(); len(af)
1
sage: af == [fn]
True
sage: reset()
sage: af = attached_files(); len(af)
1
sage: reset(attached=True)
sage: af = attached_files(); len(af)
0
sage: sage.misc.reset.EXCLUDE.remove('fn')
```

```plaintext
>>> from sage.all import *
```

```plaintext
>>> x = Integer(5)
>>> reset()

# needs sage.symbolic
x

>>> fn = tmp_filename(ext='.py')
>>> sage.misc.reset.EXCLUDE.add('fn')
>>> with open(fn, 'w') as f:
```
... _ = f.write('a = 111')
>>> attach(fn)
>>> af = attached_files(); len(af)
1
>>> af == [fn]
True
>>> reset()
>>> af = attached_files(); len(af)
1
>>> reset(attach=True)
>>> af = attached_files(); len(af)
0
>>> sage.misc.reset.EXCLUDE.remove('fn')

sage.misc.reset.reset_interfaces()

sage.misc.reset.restore(vars=None)

Restore predefined global variables to their default values.

INPUT:

- vars – string or list (default: None), if not None, restores just the given variables to the default value.

EXAMPLES:

```
sage: x = 10; y = 15/3; QQ=red
sage: QQ
'red'
sage: restore('QQ')
sage: QQ
Rational Field
sage: x
10
sage: y = var('y')  #...
←needs sage.symbolic
sage: restore('x y')
←needs sage.symbolic
x
sage: y
Traceback (most recent call last):
...
NameError: name 'y' is not defined
sage: x = 10; y = 15/3; QQ=red
sage: ww = 15
sage: restore()
sage: x, QQ, ww  #...
←needs sage.symbolic
(x, Rational Field, 15)
sage: restore('ww')
sage: ww
Traceback (most recent call last):
...
NameError: name 'ww' is not defined
```

```python
>>> from sage.all import *
>>> x = Integer(10); y = Integer(15)/Integer(3); QQ='red'
(continues on next page)
```
6.1.3 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
```

```
>>> from sage.all import *
>>> 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
```

```
>>> from sage.all import *
>>> 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')
```
Functions and classes

```
class sage.misc.viewer.Viewer

Bases: 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
old_browser = viewer.browser()  # indirect doctest
viewer.browser('open -a /Applications/Firefox.app')
viewer.browser()  # open -a /Applications/Firefox.app
viewer.browser(old_browser)  # restore old value
```
Utilities, Release 10.4

```
sage: from sage.misc.viewer import viewer
go
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

>>> from sage.all import *
>>> from sage.misc.viewer import viewer
>>> old_pdf_app = viewer.pdf_viewer()
>>> 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

>>> from sage.all import *
>>> from sage.misc.viewer import viewer
>>> old_png_app = viewer.png_viewer()
>>> 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.

EXAMPLES:

```
sage: from sage.misc.viewer import browser
sage: browser() # random -- depends on OS, etc.
'open'
```

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. '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.

6.1. Interactive Sage Sessions
EXAMPLES:

```python
sage: from sage.misc.viewer import dvi_viewer
sage: dvi_viewer()  # random -- depends on OS, etc.
'open'
```

```python
>>> from sage.all import *
>>> from sage.misc.viewer import dvi_viewer
>>> dvi_viewer()  # random -- depends on OS, etc.
'open'
```

```
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.
EXAMPLES:
```

```python
sage: from sage.misc.viewer import pdf_viewer, viewer
sage: old_pdf_app = viewer.pdf_viewer()
sage: viewer.pdf_viewer('acroread')
sage: pdf_viewer()
'acroread'
sage: viewer.pdf_viewer('old_pdf_app')
```

```python
>>> from sage.all import *
>>> from sage.misc.viewer import pdf_viewer, viewer
>>> old_pdf_app = viewer.pdf_viewer()
>>> viewer.pdf_viewer('acroread')
>>> pdf_viewer()
'acroread'
>>> 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.
EXAMPLES:
```

```python
sage: from sage.misc.viewer import png_viewer
sage: png_viewer()  # random -- depends on OS, etc.
'xdg-open'
```

```python
>>> from sage.all import *
>>> from sage.misc.viewer import png_viewer
>>> png_viewer()  # random -- depends on OS, etc.
'xdg-open'
```
6.1.4 Pager for showing strings

Currently we just use the IPython pager. If we want to use something else, we can just change this function.

Any code in sage that uses a pager should use this pager.

```
sage.misc.pager.pager()
```

6.1.5 Format Sage documentation for viewing with IPython and the notebook

AUTHORS:

- 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
- Simon King (2011-09): Use os.linesep, avoid destruction of embedding information, enable nodetex in a docstring. Consequently use sage_getdoc.

```
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 (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 \geq k. A website: sagemath.org.'

sage: detex(r"More math: x \mapsto y. (\bf Bold face).")
'More math: x |--> y. { Bold face}.'

sage: detex(r"a, b, c, \ldots, z")
'a, b, c, ..., z'

sage: detex(r"a, b, c, \ldots, z", embedded=True)
'\`a, b, c, \ldots, z'"

sage: detex(r"\left|\left(\frac{x+y}{2}\right)\right|\)"
'(| x + y |)'

sage: detex(r"\left\langle\left\{\left\{\left\{\left\{a\right\}\right\}\right\}\right\}\right\rangle")
'\langle\langle\langle\langle a\rangle\rangle\rangle\rangle'
```

6.1. Interactive Sage Sessions
```python
>>> from sage.all import *
>>> from sage.misc.sagedoc import detex
>>> detex(r'Some math: $n \geq k$. A website: \url{sagemath.org}.
')

>>> detex(r'More math: $x \mapsto y$. \{\textbf{Bold face}\}.
')
'More math: x |---> y. { Bold face}.

>>> detex(r'\alpha, \beta, c, \ldots, z\n')
'\alpha, \beta, c, ..., z

>>> detex(r'\alpha, \beta, c, \ldots, z\n', embedded=True)
'\alpha, \beta, c, \ldots, z

>>> detex(r'\left(\vert x\ast y \right\]\n')
'(| x * y |]

>>> detex(r'\left(\leq \leftarrow \rightarrow \to\n')
'(| x * y |]

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.

INPUT:

- s – string
- embedded – boolean (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().

See also:
sage.misc.sageinspect.sage_getdoc() to get the formatted documentation of a given object.

EXAMPLES:

```
If the first line of the string is ‘nodetex’, remove ‘nodetex’ but don’t modify any TeX commands:

```python
sage: format("nodetex\n\nx \\geq y")
'\text{x} \geq \text{y}'
```

Testing a string enclosed in triple angle brackets:

```python
>>> from sage.all import *
>>> format("nodetex\n\nx \\geq y")
'\text{x} \geq \text{y}'
```

This function parses `results`: each line should have either the form `FILENAME` or `FILENAME: string` where `FILENAME` is the file in which the string that matched the search was found. 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:**

6.1. Interactive Sage Sessions
Utilities, Release 10.4

```
sage: from sage.misc.sagedoc import format_search_as_html

sage: format_search_as_html('Source', 'algebras/steenrod_algebra_element.py: ... an antihomomorphism: if we call the antipode `c`, then', 'antipode...


sage: format_search_as_html('Other', 'html/en/reference/sage/algebras/steenrod_algebra_element.html: an antihomomorphism: if we call the antipode `<span class="math">c</span>`, then', 'antipode antihomomorphism')


>>> from sage.all import *

sage: from sage.misc.sagedoc import format_search_as_html

>>> format_search_as_html('Source', 'algebras/steenrod_algebra_element.py: ... an antihomomorphism: if we call the antipode `c`, then', 'antipode...


sage: format_search_as_html('Other', 'html/en/reference/sage/algebras/steenrod_algebra_element.html: an antihomomorphism: if we call the antipode `<span class="math">c</span>`, then', 'antipode antihomomorphism')


>>> from sage.all import *

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:

```python
sage: from sage.misc.sagedoc import format_src
sage: format_src('unladen swallow')
'unladen swallow'
sage: format_src('<<<Sq>>>'+ '[5:15] 
 needs sage.combinat sage.modules
'Sq('+nums'):'

>>> from sage.all import *

>>> from sage.misc.sagedoc import format_src
>>> format_src('unladen swallow')
'unladen swallow'
>>> format_src('<<<Sq>>>'+ [Integer(5):Integer(15)]
```

(continues on next page)
Utilities, Release 10.4

(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:

```python
sage: help()
Welcome to Sage ...
```

```python
>>> from sage.all import *
```

```python
>>> help()
Welcome to Sage ...
```

(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:

```python
sage: from sage.misc.sagedoc import my_getsource
sage: s = my_getsource(identity_matrix)  # needs sage.modules
s[15:34]  # needs sage.modules
'def identity_matrix'
```

```python
>>> from sage.all import *
```  

```python
>>> from sage.misc.sagedoc import my_getsource
```  

```python
>>> s = my_getsource(identity_matrix)  # needs sage.modules
>>> s[Integer(15):Integer(34)]  # needs sage.modules
'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: $x=y$')
doctest:warning...
DeprecationWarning: using dollar signs to mark up math in Sage docstrings
is deprecated; use backticks instead
See https://github.com/sagemath/sage/issues/33973 for details.
'some math: \text{`x=y'}'
```

```python
>>> from sage.all import *
>>> from sage.misc.sagedoc import process_dollars
>>> process_dollars('hello')
'hello'
```

```python
>>> process_dollars('some math: $x=y$')
doctest:warning...
DeprecationWarning: using dollar signs to mark up math in Sage docstrings
is deprecated; use backticks instead
See https://github.com/sagemath/sage/issues/33973 for details.
'some math: \text{Ax=yA}
```

Replace \$ with $, and don’t do anything when backticks are involved:

```
sage: process_dollars(r'a \``\$REAL\`` dollar sign: \$')
'a \``\$REAL\`` dollar sign: $'
```

```python
>>> from sage.all import *
>>> 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
```

```python
>>> from sage.all import *
>>> s = '\n first line\n indented $x=y$'
>>> 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}'
```

```python
>>> from sage.all import *
>>> 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}$$'
```
Utilities, Release 10.4

>>> from sage.all import *

>>> 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 :issue:`NUM` with https://github.com/sagemath/sage/issues/NUM, and similarly with :python:`TEXT` and :wikipedia:`TEXT`, looking up the url from the dictionary extlinks in sage_docbuild.conf. 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 (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 :issue:`1234` for more information.', embedded=True)
'See :issue:`1234` for more information.'

sage: process_extlinks('see :python:`Implementing Descriptors <reference/datamodel.html#implementing-descriptors>` ...')
'see https://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

6.1. Interactive Sage Sessions 585
This function is called by \texttt{\textit{format()}}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.misc.sagedoc import process_mathtt
sage: process_mathtt(r'\texttt{\textbackslash e^\texttt{self}}')
e^self
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> from sage.misc.sagedoc import process_mathtt
>>> process_mathtt(r'\texttt{\textbackslash e^\texttt{self}}')
e^self
\end{verbatim}

\begin{verbatim}
sage.misc.sagedoc.process_optional_doctest_tags(s)
\end{verbatim}

Remove \# optional/needs doctest tags for present features from docstring \textit{s}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.misc.sagedoc import process_optional_doctest_tags sage: process_optional_doctest_tags("sage: \# needs sage.rings.finite_rings:sage: K.<x> = FunctionField(GF(5^2,'a')); 
KnRational function field in x over Finite Field in a of size 5^2") # needs sage.rings.finite_rings "sage: K.<x> = FunctionField(GF(5^2,'a')); KnRational function field in x over Finite Field in a of size 5^2"
\end{verbatim}

\begin{verbatim}
sage.misc.sagedoc.search_def(name, extra1='', extra2='', extra3='', extra4='', extra5='', **kwds)
\end{verbatim}

Search Sage library source code for function definitions containing \textit{name}. The search is case-insensitive by default.

\textbf{INPUT:} same as for \texttt{\textit{search_src()}}.

\textbf{OUTPUT:} same as for \texttt{\textit{search_src()}}.

\textbf{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.

\textbf{EXAMPLES:}

See the documentation for \texttt{\textit{search_src()}} for more examples.

\begin{verbatim}
sage: print(search_def("fetch", interact=\texttt{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="\texttt{pyx}\", interact=\texttt{False})) # random \# long time
matrix/matrix0.pyx:  cdef fetch(self, key):
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> print(search_def("fetch", interact=\texttt{False})) # random \# long time
matrix/matrix0.pyx:  cdef fetch(self, key):
matrix/matrix0.pxd:  cdef fetch(self, key)
>>> print(search_def("fetch", path_re="\texttt{pyx}\", interact=\texttt{False})) # random \# long time
matrix/matrix0.pyx:  cdef fetch(self, key):
\end{verbatim}

\begin{verbatim}
sage.misc.sagedoc.search_doc(string, extra1="", extra2="", extra3="", extra4="", extra5="", **kwds)
\end{verbatim}

Search Sage HTML documentation for lines containing \textit{string}. The search is case-insensitive by default.
The file paths in the output are relative to $SAGE_DOC.

INPUT: same as for search_src().

OUTPUT: same as for search_src().

EXAMPLES:

See the documentation for search_src() for more examples.

```python
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 (the string)
```

```python
>>> from sage.all import *
>>> 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 (the string)
```

If you search the documentation for 'tree', then you will get too many results, because many lines in the doc-
umentation 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.

```python
sage: # long time, needs sagemath_doc_html
sage: N = len(search_doc('tree', interact=False).splitlines())
sage: L = search_doc('tree', whole_word=True, interact=False).splitlines()
sage: len(L) < N
True
sage: import re
sage: tree_re = re.compile(r'\^\Wtree\W\$)', re.I)
sage: all(tree_re.search(l) for l in L)
True
```

```python
>>> from sage.all import *
>>> # long time, needs sagemath_doc_html
>>> N = len(search_doc('tree', interact=False).splitlines())
>>> L = search_doc('tree', whole_word=True, interact=False).splitlines()
>>> len(L) < N
True
>>> import re
>>> tree_re = re.compile(r'\^\Wtree\W\$)', re.I)
>>> all(tree_re.search(l) for l in L)
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-insensitive by default.

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 (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** (default: True) – if False, perform a case-sensitive search

- **multiline** (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** (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** (default: ") – regular expression which the filename (including the path) must match.

- **module** (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-insensitive unless **ignore_case** is False.

---

**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](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
```

```
>>> from sage.all import *
>>> 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
```

```
>>> from sage.all import *
>>> 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:
The following produces an error because the string ‘fetch’ is a malformed regular expression:

```python
sage: print(search_src(" fetch\(", "def", interact=False))
Traceback (most recent call last):
  ... error: missing ), unterminated subpattern at position 6
```

To fix this, escape the parenthesis with a backslash:

```python
sage: print(search_src(r" fetch\(", "def", interact=False))  # random # long time
```

As noted above, the search is case-insensitive, but you can make it case-sensitive with the ‘ignore_case’ key word:

```python
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
True
sage: s = search_src('MatRiX', path_re='matrix', interact=False, ignore_case=False); s.find('x') > 0
False
```
>>> from sage.all import *
... s = search_src('Matrix', path_re='matrix', interact=False); s.find('x') >
... → Integer(0)
True

>>> s = search_src('MatRiX', path_re='matrix', interact=False); s.find('x') >
... → Integer(0)
True

>>> s = search_src('MatRiX', path_re='matrix',
... ... interact=False, ignore_case=False); s.find('x') > Integer(0)
False

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()) < 40
True
sage: len(search_src('log', 'derivative',
... ... interact=False, multiline=True).splitlines()) > 70
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(r'\*sage\[:\] .*search_src\(', interact=False)) # long time
misc/sagedoc.py:... len(search_src("matrix", interact=False).splitlines())...
misc/sagedoc.py:... len(search_src("matrix", module="sage.calculus",
... → interact=False).splitlines())...
misc/sagedoc.py:... len(search_src("matrix", path_re="calc"...
misc/sagedoc.py:... print(search_src(" fetch\(", "def", interact=False))...
misc/sagedoc.py:... print(search_src(" fetch\(", "def", interact=False))...
misc/sagedoc.py:... print(search_src(" fetch\(", "def", "pyx", interact=False))..
... →.
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',.
misc/sagedoc.py:... len(search_src('log', 'derivative', interact=False).
... →splitlines()) < 40...
misc/sagedoc.py:... len(search_src('log', 'derivative'...
misc/sagedoc.py:... print(search_src(r'\*sage\[:\] .*search_src\(',
... → interact=False))...
misc/sagedoc.py:... len(search_src("matrix", interact=False).splitlines()) > 9000.
... →
misc/sagedoc.py:... print(search_src("matrix", 'column', 'row', 'sub',...
(continues on next page)
sage.misc.sagedoc.skip_TESTS_block (docstring)

Remove blocks labeled “TESTS:” from docstring.

INPUT:

• docstring, a string

A “TESTS” block is a block starting “TESTS:” (or the same with two colons), on a line on its own, and ending either with a line indented less than “TESTS”, or with a line with the same level of indentation – not more – matching one of the following:

• a Sphinx directive of the form “.. foo:”, optionally followed by other text.

• text of the form “UPPERCASE:”, optionally followed by other text.

• lines which look like a reST header: one line containing anything, followed by a line consisting only of a string of hyphens, equal signs, or other characters which are valid markers for reST headers: - = ` : ! " ^ * + # < >. However, lines only containing double colons :: do not end “TESTS” blocks.

Return the string obtained from docstring by removing these blocks.

EXAMPLES:

```python
sage: from sage.misc.sagedoc import skip_TESTS_block
sage: start = ' Docstring


sage: test = ' TESTS: \nHere is a test::

sage: test2 = ' TESTS: \n
sage: from sage.all import *
>>> from sage.misc.sagedoc import skip_TESTS_block
>>> start = ' Docstring

```
Utilities, Release 10.4

(continued from previous page)

```python
>>> test = ' TESTS: 
Here is a test::
sage: 2+2 
5 

>>> test2 = ' TESTS:: 
sage: 2+2 
6 

Test lines starting with “REFERENCES”:```

```python
sage: refs = ' REFERENCES: 

sage: skip_TESTS_block(start + test + refs).rstrip() == (start + refs).rstrip()
True

sage: skip_TESTS_block(start + test + test2 + refs).rstrip() == (start + refs).
→rstrip()
True

sage: skip_TESTS_block(start + test + refs + test2).rstrip() == (start + refs).
→rstrip()
True
```

```python
>>> from sage.all import *

refs = ' REFERENCES:

```

```python
>>> from sage.all import *

```

Test Sphinx directives:

```python
sage: directive = '.. todo:: 
do some stuff 

sage: skip_TESTS_block(start + test + refs + test2 + directive).rstrip() ==
→(start + refs + directive).rstrip()
True
```

```python
>>> from sage.all import *

```

Test unindented lines:

```python
sage: unindented = 'NOT INDENTED

sage: skip_TESTS_block(start + test + unindented).rstrip() == (start +
→unindented).rstrip()
True

sage: skip_TESTS_block(start + test + unindented + test2 + unindented).rstrip()...
→== (start + unindented + unindented).rstrip()
True
```

```python
>>> from sage.all import *


unindented = 'NOT INDENTED

```

(continues on next page)
Test headers:

```python
def skip_TESTS_block(start + test + header): return start + header
```

Not a header because the characters on the second line must all be the same:

```python
def skip_TESTS_block(start + test + fake_header): return start
```

Not a header because it's indented compared to 'TEST' in the string test:

```python
def skip_TESTS_block(start + test + another_fake):
```

Double colons :: are also not considered as headers (Issue #27896):

```python
def skip_TESTS_block(start + test2 + colons):
```

6.1. Interactive Sage Sessions 593
6.1.6 Sphinx configuration shared by sage.misc.sphinxify and sage_docbuild

```python
class sage.misc.sagedoc_conf.SagemathTransform(document, startnode=None)
    Bases: Transform
    Transform for code-blocks.
    This allows Sphinx to treat code-blocks with prompt “sage:” as associated with the pycon lexer, and in particular, to change “<BLANKLINE>” to a blank line.

    apply()

    default_priority = 500
    Numerical priority of this transform, 0 through 999 (override).

sage.misc.sagedoc_conf.process_directives(app, what, name, obj, options, docstringlines)
    Remove ‘nodetex’ and other directives from the first line of any docstring where they appear.

sage.misc.sagedoc_conf.process_docstring_aliases(app, what, name, obj, options, docstringlines)
    Change the docstrings for aliases to point to the original object.

sage.misc.sagedoc_conf.process_docstring_cython(app, what, name, obj, options, docstringlines)
    Remove Cython’s filename and location embedding.

sage.misc.sagedoc_conf.process_docstring_module_title(app, what, name, obj, options, docstringlines)
    Removes the first line from the beginning of the module’s docstring. This corresponds to the title of the module’s documentation page.

sage.misc.sagedoc_conf.process_dollars(app, what, name, obj, options, docstringlines)
    Replace dollar signs with backticks.
    See sage.misc.sagedoc.process_dollars for more information.

sage.misc.sagedoc_conf.process_inherited(app, what, name, obj, options, docstringlines)
    If we’re including inherited members, omit their docstrings.

sage.misc.sagedoc_conf.setup(app)

sage.misc.sagedoc_conf.skip_TESTS_block(app, what, name, obj, options, docstringlines)
    Skip blocks labeled “TESTS:”.
    See sage.misc.sagedoc.skip_TESTS_block for more information.
```

6.1.7 Process docstrings with Sphinx

Processes docstrings with Sphinx. Can also be used as a commandline script:

```
python sphinxify.py <text>
```

AUTHORS:

• Tim Joseph Dumol (2009-09-29): initial version

```
sage.misc.sphinxify.sphinxify (docstring, format='html')
    Runs Sphinx on a docstring, and outputs the processed documentation.
```

INPUT:
• docstring – string: a ReST-formatted docstring
• format – string (default: ‘html’); either ‘html’ or ‘text’

OUTPUT:
• string – Sphinx-processed documentation, in either HTML or plain text format, depending on the value of format

EXAMPLES:

```python
sage: from sage.misc.sphinxify import sphinxify
sage: sphinxify('A test')
'<div class="docstring">

<p>A test</p>
</div>'
sage: sphinxify('**Testing**\n\nmonospace')
'<div class="docstring">...<strong>Testing</strong><span class="math...</p>
</div>'
sage: sphinxify('x=y')
'<div class="docstring">

<p>x=y</p>
</div>'
sage: sphinxify('x=y', format='text')
'x=y
'
sage: sphinxify(':math:x=y', format='text')
'x=y
'

>>> from sage.all import *
>>> from sage.misc.sphinxify import sphinxify
>>> sphinxify('A test')
'<div class="docstring">

<p>A test</p>
</div>'

>>> sphinxify('**Testing**\n\nmonospace')
'<div class="docstring">...<strong>Testing</strong><span class="math...</p>
</div>'

>>> sphinxify('x=y')
'<div class="docstring">

<p>x=y</p>
</div>'

>>> sphinxify('x=y', format='text')
'x=y
'

>>> sphinxify(':math:x=y', format='text')
'x=y
'
```

6.2 Distribution

6.2.1 Listing Sage packages

This module can be used to see which Sage packages are installed and which packages are available for installation.

For more information about creating Sage packages, see the “Packaging Third-Party Code” section of the Sage Developer’s Guide.

Actually installing the packages should be done via the command line, using the following commands:

• `sage -i PACKAGE_NAME` – install the given package
• `sage -f PACKAGE_NAME` – re-install the given package, even if it was already installed

To list the packages available, either use in a terminal one of `sage -standard`, `sage -optional` or `sage -experimental`. Or the following command inside Sage:
sage: from sage.misc.package import list_packages
sage: pkgs = list_packages(local=True)  # optional - sage_spkg
sage: sorted(pkgs.keys())  # optional - sage_spkg, random
['4ti2',
 'alabaster',
 ...
 'zlib']

>>> from sage.all import *
>>> from sage.misc.package import list_packages
>>> pkgs = list_packages(local=True)  # optional - sage_spkg
>>> sorted(pkgs.keys())  # optional - sage_spkg, random
['4ti2',
 'alabaster',
 ...
 'zlib']

## Functions

**class** sage.misc.package.PackageInfo**(name: str, type: str | None = None, source: str | None = None, installed_version: str | None = None, remote_version: str | None = None)**

Bases: NamedTuple

Represents information about a package.

**installed_version**: str | None

Alias for field number 3

**is_installed**()

Whether the package is installed in the system.

**name**: str

Alias for field number 0

**remote_version**: str | None

Alias for field number 4

**source**: str | None

Alias for field number 2

**type**: str | None

Alias for field number 1

**sage.misc.package.installed_packages**(exclude_pip=True)**

Return a dictionary of all installed packages, with version numbers.

**INPUT:**

- exclude_pip = (default: True) whether “pip” packages are excluded from the list

**EXAMPLES:**

Below we test for a standard package without spkg-configure.m4 script that should be installed in SAGE_LOCAL. When Sage is installed by the Sage distribution (indicated by feature sage_spkg), we should have the installation record for this package. (We do not test for installation records of Python packages. Our SAGE_VENV is not necessarily the main Sage venv; it could be a user-created venv or a venv created by tox.):
Utilities, Release 10.4

```python
sage: from sage.misc.package import is_package_installed
sage: is_package_installed('conway_polynomials')  # optional - sage_spkg
True
```

Giving just the beginning of the package name is not good enough:

```python
sage: is_package_installed('conway_poly')  # optional - sage_spkg
False
```

Otherwise, installing “pillow” would cause this function to think that “pil” is installed, for example.

**Note:** Do not use this function to check whether you can use a feature from an external library. This only checks whether something was installed with `sage -i` but it may have been installed by other means (for example if this copy of Sage has been installed as part of a distribution.) Use the framework provided by `sage.features` to check whether a library is installed and functional.
sage.misc.package.\texttt{is\_package\_installed\_and\_updated}(\texttt{package})

Return whether the given package is installed and up-to-date.

**INPUT:**

- \texttt{package} – the name of the package.

**EXAMPLES:**

```sage
from sage.misc.package import is_package_installed_and_updated
sage: is_package_installed_and_updated("alabaster")  # optional - build, random
False
```

```sage
from sage.all import *
from sage.misc.package import is_package_installed_and_updated
>> is_package_installed_and_updated("alabaster")  # optional - build, random
False
```

sage.misc.package.\texttt{list\_packages}(\texttt{pkg\_sources}, \texttt{local}, \texttt{ignore\_URLError}, \texttt{exclude\_pip}, \*\texttt{pkg\_types})

Return a dictionary of information about each package.

The keys are package names and values are named tuples with the following keys:

- 'type': either 'base', 'standard', 'optional', or 'experimental'
- 'source': either 'normal', `pip', or 'script'
- 'installed': boolean
- 'installed\_version': \texttt{None} or a \texttt{string}
- 'remote\_version': \texttt{string}

**INPUT:**

- \texttt{pkg\_types} – (optional) a sublist of 'base', 'standard', 'optional', or 'experimental'. If provided, list only the packages with the given type(s), otherwise list all packages.
- \texttt{pkg\_sources} – (optional) a sublist of 'normal', `pip', or 'script'. If provided, list only the packages with the given source(s), otherwise list all packages.
- \texttt{local} – (default: False) if set to True, then do not consult remote (PyPI) repositories for package versions (only applicable for 'pip' type)
- \texttt{exclude\_pip} – (default: False) if set to True, then pip packages are not considered. This is the same as removing 'pip' from pkg\_sources.
- \texttt{ignore\_URLError} – (default: False) if set to True, then connection errors will be ignored

**EXAMPLES:**

```sage
# optional - sage_spkg
from sage.misc.package import list_packages
L = list_packages('standard')
sorted(L.keys())  # random
['alabaster',
 'babel',
 ...]
sage_conf_info = L['sage_conf']
sage: sage_conf_info.type
'standard'
```
Utilities, Release 10.4

(continued from previous page)

```python
sage: sage_conf_info.is_installed()
True
sage: sage_conf_info.source
'script'

sage: # optional - sage_spkg internet
sage: L = list_packages(pkg_sources=['pip'], local=True)
sage: bp_info = L['biopython']
sage: bp_info.type
'optional'
sage: bp_info.source
'pip'
```

```python
>>> from sage.all import *
>>> from sage.misc.package import list_packages
>>> L = list_packages('standard')
>>> sorted(L.keys())  # random
['alabaster',
 'babel',
 '...' 
 'zlib']

>>> sage_conf_info = L['sage_conf']

>>> sage_conf_info.type
'standard'

>>> sage_conf_info.is_installed()
True

>>> sage_conf_info.source
'script'

>>> # optional - sage_spkg internet
>>> L = list_packages(pkg_sources=['pip'], local=True)

>>> bp_info = L['biopython']

>>> bp_info.type
'optional'

>>> bp_info.source
'pip'
```

Check the option exclude_pip:

```python
sage: [p for p, d in list_packages('optional', exclude_pip=True).items()  #...
    optional - sage_spkg
    ....: if d.source == 'pip']
[]
```

```python
>>> from sage.all import *

>>> [p for p, d in list_packages('optional', exclude_pip=True).items()  #...
    optional - sage_spkg
    ... if d.source == 'pip']
[]
```

```
sage.misc.package.package_manifest(package)

Return the manifest for package.

INPUT:

• package -- package name
```
The manifest is written in the file SAGE_SPKG_INST/package-VERSION. It is a JSON file containing a dictionary with the package name, version, installation date, list of installed files, etc.

EXAMPLES:

```python
sage: from sage.misc.package import package_manifest
sage: manifest = package_manifest('conway_polynomials')
True
sage: manifest['package_name'] == 'conway_polynomials'
True
sage: 'files' in manifest
True
```

Test a nonexistent package:

```python
sage: package_manifest('dummy-package')  # optional - sage_spkg
Traceback (most recent call last):
  ...  
KeyError: 'dummy-package'
```

```
>>> from sage.all import *
...  # optional - sage_spkg
>>> package_manifest('dummy-package')  # optional - sage_spkg
Traceback (most recent call last):
  ...  
KeyError: 'dummy-package'
```

`sage.misc.package.package_versions` *(package_type, local=False)*

Return version information for each Sage package.

INPUT:

- `package_type` — (string) one of "standard", "optional" or "experimental"
- `local` — (boolean, default: False) only query local data (no internet needed)

For packages of the given type, return a dictionary whose entries are of the form 'package': (installed, latest), where installed is the installed version (or None if not installed) and latest is the latest available version. If the package has a directory in SAGE_ROOT/build/pkgs/, then latest is determined by the file package-version.txt in that directory. If `local` is False, then Sage’s servers are queried for package information.

See also:

`sage.misc.package.list_packages()`

EXAMPLES:

```python
sage: from sage.misc.package import package_versions
sage: std = package_versions('standard', local=True)
(continues on next page)`
sage: 'gap' in std
True
sage: std['zlib']  # random
('1.2.11.p0', '1.2.11.p0')

>>> from sage.all import *
>>> # optional - sage_spkg
>>> from sage.misc.package import package_versions
>>> std = package_versions('standard', local=True)

True

>>> std['zlib']  # random
('1.2.11.p0', '1.2.11.p0')

sage.misc.package.pip_installed_packages(normalization=None)

Return a dictionary name -> version of installed pip packages.

This command returns all pip-installed packages. Not only Sage packages.

INPUT:

• normalization = (default: None) according to which rule to normalize the package name, either None (as is) or 'spkg' (format as in the Sage distribution in build/pkgs/), i.e., lowercased and dots and dashes replaced by underscores.

EXAMPLES:

sage: # optional - sage_spkg
sage: from sage.misc.package import pip_installed_packages
sage: d = pip_installed_packages()

scipy in d or SciPy in d
# needs scipy
True

beautifulsoup4 in d
# needs beautifulsoup4
True

prompt-toolkit in d or prompt_toolkit in d
# whether - or _ appears in
# the name depends on the setuptools version used for building the package
True

d = pip_installed_packages(normalization='spkg')

'...
...'

>>> from sage.all import *
>>> # optional - sage_spkg
>>> from sage.misc.package import pip_installed_packages

'scipy' in d or 'SciPy' in d
# needs scipy
True

'beautifulsoup4' in d
# needs beautifulsoup4
True

'prompt-toolkit' in d or 'prompt_toolkit' in d
# whether - or _ appears in
# the name depends on the setuptools version used for building the package
True

(continues on next page)
```python
>>> d = pip_installed_packages(normalization='spkg')
>>> d['prompt_toolkit']
'...
>>> d['scipy']

needs scipy
'...
```

**sage.misc.package.pip_remote_version**(pkg, pypi_url='https://pypi.org/pypi', ignore_URLError=False)

Return the version of this pip package available on PyPI.

**INPUT:**

- `pkg` – the package
- `pypi_url` – (string, default: standard PyPI url) an optional Python package repository to use
- `ignore_URLError` – (default: False) if set to True then no error is raised if the connection fails and the function returns None

**EXAMPLES:**

The following test does fail if there is no TLS support (see e.g. Issue #19213):

```python
sage: from sage.misc.package import pip_remote_version
sage: pip_remote_version('beautifulsoup4')  # optional - internet # not tested
'...
```

These tests are reliable since the tested package does not exist:

```python
sage: nap = 'hey_this_is_NOT_a_python_package'
sage: pypi = 'http://this.is.not.pypi.com/
sage: pip_remote_version(nap, pypi_url=pypi, ignore_URLError=True)  # optional -

internet
```

```
from sage.all import *
```
```python
from sage.misc.package import pip_remote_version
```
Traceback (most recent call last):
...
HTTPError: HTTP Error 404: Not Found

sage.misc.package.pkgname_split(name)

Split a pkname into a list of strings, 'name, version'.
For some packages, the version string might be empty.

EXAMPLES:

sage: from sage.misc.package import pkgname_split
sage: pkgname_split('hello_world-1.2')
['hello_world', '1.2']

>>> from sage.all import *
>>> from sage.misc.package import pkgname_split
>>> pkgname_split('hello_world-1.2')
['hello_world', '1.2']

sage.misc.package.spkg_type(name)

Return the type of the Sage package with the given name.

INPUT:

* name – string giving the subdirectory name of the package under SAGE_PKGS

EXAMPLES:

sage: from sage.misc.package import spkg_type
sage: spkg_type('pip')
'standard'  # optional - sage_spkg

>>> from sage.all import *
>>> from sage.misc.package import spkg_type
>>> spkg_type('pip')
'standard'  # optional - sage_spkg

OUTPUT:

The type as a string in ('base', 'standard', 'optional', 'experimental'). If no SPKG exists with the given name (or the directory SAGE_PKGS is not available), None is returned.

6.3 Credits

6.3.1 Dependency usage tracking for citations

sage.misc.citation.cython_profile_enabled()

Return whether Cython profiling is enabled.

EXAMPLES:
Utilities, Release 10.4

```python
sage: from sage.misc.citation import cython_profile_enabled
sage: cython_profile_enabled()  # random
False
sage: type(cython_profile_enabled()) is bool
True
```

```python
>>> from sage.all import *
>>> from sage.misc.citation import cython_profile_enabled
>>> cython_profile_enabled()  # random
False
>>> type(cython_profile_enabled()) is bool
True
```

```python
sage.misc.citation.get_systems(cmd)
```

Return a list of the systems used in running the command `cmd`.

Note that the results can sometimes include systems that did not actually contribute to the computation. Due to caching, it could miss some dependencies as well.

**INPUT:**

- `cmd` – a string to run

**Warning:** In order to properly support Cython code, this requires that Sage was compiled with the environment variable `SAGE_PROFILE=yes`. If this was not the case, a warning will be given when calling this function.

**EXAMPLES:**

```python
sage: from sage.misc.citation import get_systems
sage: get_systems('print("hello")')  # random (may print warning)
[]
sage: integrate(x^2, x)  # Priming coercion model
1/3*x^3
sage: get_systems('integrate(x^2, x)')
['Maxima', 'ginac']
sage: R.<x,y,z> = QQ[]
sage: I = R.ideal(x^2+y^2, z^2+y)
sage: get_systems(I.primary_decomposition())
['Singular']
```

```python
>>> from sage.all import *
>>> from sage.misc.citation import get_systems
>>> get_systems('print("hello")')  # random (may print warning)
[]
>>> integrate(x**Integer(2), x)  # Priming coercion model
1/3*x^3
>>> get_systems('integrate(x^2, x)')
['Maxima', 'ginac']
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> I = R.ideal(x**Integer(2)+y**Integer(2), z**Integer(2)+y)
>>> get_systems(I.primary_decomposition())
['Singular']
```
6.3.2 License

class sage.misc.copying.License
  Bases: object
7.1 Testing

7.1.1 Unit testing for Sage objects

class sage.misc.sage_unittest.InstanceTester(instance, elements=None, verbose=False, prefix='', max_runs=4096, max_samples=None, **options)

Bases: TestCase

A gadget attached to an instance providing it with testing utilities.

EXAMPLES:

```python
sage: from sage.misc.sage_unittest import InstanceTester
sage: InstanceTester(instance = ZZ, verbose = True, elements = [1,2,3])
Testing utilities for Integer Ring
```

```python
>>> from sage.all import *
>>> from sage.misc.sage_unittest import InstanceTester
>>> InstanceTester(instance = ZZ, verbose = True, elements = [Integer(1), Integer(2),Integer(3)])
Testing utilities for Integer Ring
```

This is used by SageObject._tester, which see:

```python
sage: QQ._tester()
Testing utilities for Rational Field
```

```python
>>> from sage.all import *
>>> QQ._tester()
Testing utilities for Rational Field
```

info (message, newline=True)

Display user information

EXAMPLES:

```python
sage: from sage.misc.sage_unittest import InstanceTester
sage: tester = InstanceTester(ZZ, verbose = True)
```

```python
sage: tester.info("hello"); tester.info("world")
```

(continues on next page)
```python
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")
```

```bash
>>> from sage.all import *
>>> from sage.misc.sage_unittest import InstanceTester
>>> tester = InstanceTester(ZZ, verbose = True)
>>> tester.runTest()
>>> from sage.all import *
>>> from sage.misc.sage_unittest import InstanceTester
>>> tester = InstanceTester(ZZ, verbose = True)
>>> tester.runTest()
```

```python
longMessage = False
```

```python
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:**

```python
sage: from sage.misc.sage_unittest import InstanceTester
sage: tester = InstanceTester(ZZ, verbose = True)
sage: tester.runTest()
```

```python
some_elements(S=None, repeat=None)
```

Return a list (or iterable) of elements of the instance 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.
- repeat – integer (default: None). If given, instead returns a list of tuples of length repeat from S.

**OUTPUT:**

A list of at most self._max_runs elements of S^r, or a sample of at most self._max_samples if that is not None.

**EXAMPLES:**
By default, this calls `some_elements()` on the instance:

```python
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]
```

```python
>>> from sage.all import *
>>> from sage.misc.sage_unittest import InstanceTester
>>> class MyParent(Parent):
    ...     def some_elements(self):
    ...
(continues on next page)```
...    return [Integer(1), Integer(2), Integer(3), Integer(4), Integer(5)]
...

>>> tester = InstanceTester(MyParent())
>>> list(tester.some_elements())
[1, 2, 3, 4, 5]

>>> tester = InstanceTester(MyParent(), max_runs=Integer(3))
>>> list(tester.some_elements())
[1, 2, 3]

>>> tester = InstanceTester(MyParent(), max_runs=Integer(7))
>>> list(tester.some_elements())
[1, 2, 3, 4, 5]

>>> tester = InstanceTester(MyParent(), elements=[Integer(1), Integer(3), Integer(5)])
>>> list(tester.some_elements())
[1, 3, 5]

>>> tester = InstanceTester(MyParent(), elements=[Integer(1), Integer(3), Integer(5)], max_runs=Integer(2))
>>> list(tester.some_elements())
[1, 3]

>>> tester = InstanceTester(FiniteEnumeratedSet(['a', 'b', 'c', 'd']), max_runs=Integer(3))
>>> tester.some_elements()
['a', 'b', 'c']

>>> tester = InstanceTester(FiniteEnumeratedSet([]))
>>> list(tester.some_elements())
[]

>>> tester = InstanceTester(ZZ)
>>> ZZ.some_elements() # yikes, shamelessly trivial ...
<generator object ..._some_elements_from_iterator at 0x...>

>>> list(tester.some_elements())
[0, 1, -1, 2, -2, ..., 49, -49, 50]

>>> tester = InstanceTester(ZZ, elements = ZZ, max_runs=Integer(5))
>>> list(tester.some_elements())
[0, 1, -1, 2, -2]

>>> tester = InstanceTester(ZZ, elements = srange(Integer(100)), max_runs=Integer(5))
>>> list(tester.some_elements())
[0, 1, 2, 3, 4]

>>> tester = InstanceTester(ZZ, elements = srange(Integer(3)), max_runs=Integer(5))
>>> list(tester.some_elements())
[0, 1, 2]

The `repeat` keyword can give pairs or triples from $S$:

```
sage: list(tester.some_elements(repeat=2))
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1)]
```
You can use `max_samples` to sample at random, instead of in order:

```python
sage: tester = InstanceTester(ZZ, elements = srange(8), max_samples = 4)
sage: all(t in srange(8) for t in tester.some_elements())
True
sage: all(s in srange(8) and t in srange(8) for s,t in tester.some_elements(repeat=2))
True
```
class sage.misc.sage_unittest.PythonObjectWithTests(instance)

    Bases: object

    Utility class for running basic tests on a plain Python object (that is not in SageObject). More test methods can be added here.

    EXAMPLES:

        sage: TestSuite("bla").run()

        >>> from sage.all import *
        >>> TestSuite("bla").run()

class sage.misc.sage_unittest.TestSuite(instance)

    Bases: object

    Test suites for Sage objects.

    EXAMPLES:

        sage: TestSuite(ZZ).run()

        >>> from sage.all import *
        >>> 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(overwrite = True)
        running ._test_category() . . . pass
        running ._test_eq() . . . pass
        running ._test_new() . . . pass
        running ._test_nonzero_equal() . . . pass
        running ._test_not_implemented_methods() . . . pass
        running ._test_pickling() . . . pass

        >>> from sage.all import *
        >>> TestSuite(Integer(1)).run(overwrite = True)
        running ._test_category() . . . pass
        running ._test_eq() . . . pass
        running ._test_new() . . . 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(overwrite = True)
        running ._test_an_element() . . . pass

(continues on next page)
>>> from sage.all import *
>>> S = FiniteSemigroups().example(alphabet = ('a', 'b'))
>>> TestSuite(S).run( verbose = True)
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . pass
  running ._test_new() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass

running ._test_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . pass
  running ._test_new() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass

The different test methods can be called independently:
Utilities, Release 10.4

```python
sage: S._test_associativity()
```

```python
>>> from sage.all import *
>>> 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 independently: 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:

```python
sage: S._test_associativity(elements = S)
```

```python
>>> from sage.all import *
>>> 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:

```python
sage: TestSuite(int(1)).run(verbosity = True)
running ._test_new() . . . pass
running ._test_pickling() . . . pass
```

```python
>>> from sage.all import *
>>> TestSuite(int(Integer(1))).run(verbosity = True)
running ._test_new() . . . pass
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 `CommutativeSemi-`
groups() 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:

```python
sage: TestSuite(ZZ).run()
```

```python
>>> from sage.all import *
>>> TestSuite(ZZ).run()
```

We now use the verbose option:

```python
sage: TestSuite(1).run(verbosetrue=True)
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_nonzero_equal() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
```

```python
>>> from sage.all import *
>>> TestSuite(Integer(1)).run(verbosetrue=True)
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . 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:

```python
sage: TestSuite(1).run(verbosetrue=True,skip="_test_pickling")
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_nonzero_equal() . . . pass
running ._test_not_implemented_methods() . . . pass
```

```python
sage: TestSuite(1).run(verbosetrue=True,skip=["_test_pickling","_test_˓
category"])
running ._test_eq() . . . pass
running ._test_new() . . . 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(verbos
Utilities, Release 10.4

7.1. Testing

```
>>> from sage.all import *

>>> 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()

>>> 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

>>> TestSuite(Blah()).run(verbos = True)
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
```

(continues on next page)
...PicklingError: Can't pickle <class '__main__.Blah'>: attribute lookup ...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:

```python
sage: TestSuite(Blah()).run(catch=False)
Traceback (most recent call last):
  ... File ..., in _test_b
def _test_b(self, tester): tester.fail()
  ...
AssertionError: None
```

```python
>>> from sage.all import *
>>> TestSuite(Blah()).run(catch=False)
Traceback (most recent call last):
  ... File ..., in _test_b
def _test_b(self, tester): tester.fail()
  ...
AssertionError: None
```

In conjunction with %pdb on, this allows for the debugger to jump directly to the first failure location.

```python
exception sage.misc.sage_unittest.TestSuiteFailure
Bases: AssertionError
```

```
sage.misc.sage_unittest.instance_tester (instance, tester=None, **options)
```

Return a gadget attached to instance providing testing utilities.

**EXAMPLES:**

```python
sage: from sage.misc.sage_unittest import instance_tester
dsage: tester = instance_tester(ZZ)
```

```python
sage: tester.assertTrue(1 == 1)
sage: tester.assertTrue(1 == 0)
Traceback (most recent call last):
  ...AssertionError: False is not true
```

```python
sage: tester.assertEqual(1, 1)
sage: tester.assertEqual(1, 0)
Traceback (most recent call last):
  ...AssertionError: 1 != 0
```
Utilities, Release 10.4

```python
>>> from sage.all import *
>>> from sage.misc.sage_unittest import instance_tester

>>> tester = instance_tester(ZZ)

>>> tester.assertTrue(Integer(1) == Integer(1))
>>> tester.assertTrue(Integer(1) == Integer(0))
Traceback (most recent call last):
  ...  
AssertionError: False is not true

>>> tester.assertTrue(Integer(1) == Integer(0), "this is expected to fail")
Traceback (most recent call last):
  ...  
AssertionError: this is expected to fail

>>> tester.assertEqual(Integer(1), Integer(1))
>>> tester.assertEqual(Integer(1), Integer(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).

### 7.1.2 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:

```python
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:

```python
for _ in range(10^10): test_foo(100)
```

instead of:

```python
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:

```python
while True: test_foo(print_seed=True)
```

and look at the last seed that was printed before it crashed.

`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:**

```python
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
```

```python
>>> from sage.all import *
>>> from sage.misc.random_testing import test_add_commutes
>>> test_add_commutes(Integer(2), verbose=True, seed=Integer(0))
a == -4, b == 0 ...
Passes!
a == -1/2, b == -1/95 ...
Passes!
>>> test_add_commutes(Integer(10))
>>> test_add_commutes(Integer(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

>>> from sage.all import *
>>> 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()

```  
```
>>> from sage.all import *
>>> test_add_is_mul(Integer(2), verbose=True, seed=Integer(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()

```  
```
>>> from sage.all import *
>>> test_add_is_mul(Integer(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
```

(continues on next page)
7.1.3 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:

```python
from sage.misc.test_nested_class import TestParent1, TestParent2, TestParent3,...
>>> P = TestParent1()
>>> 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:

```python
P = TestParent2()
Traceback (most recent call last):
...
TypeError: metaclass conflict: the metaclass of a derived class must be a (non-strict) subclass of the metaclasses of all its bases
TestSuite(P).run() # not tested
```
... TypeError: metaclass conflict: the metaclass of a derived class must be a (non- →strict) subclass of the metaclasses of all its bases
>>> 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()
```

```
>>> from sage.all import *

>>> P = TestParent3()

>>> TestSuite(P).run()
```

This is what all Sage’s parents using categories currently do. An alternative is to use ClasscallMetaclss as metaclass:

```
sage: P = TestParent4()
sage: TestSuite(P).run()
```

```
>>> from sage.all import *

>>> P = TestParent4()

>>> TestSuite(P).run()
```

### 7.2 Benchmarking and Profiling

#### 7.2.1 Benchmarks

```
sage.misc.benchmark.bench0()
```

Run a benchmark.

BENCHMARK:

```
sage: from sage.misc.benchmark import *
sage: print(bench0()[0])
```

Benchmark 0: Factor the following polynomial over
the rational numbers: (x^97+19*x+1)*(x^103-19*x^97+14)*(x^100-1)

```
>>> from sage.all import *

>>> from sage.misc.benchmark import *

>>> print(bench0()[Integer(0)])
```

Benchmark 0: Factor the following polynomial over
the rational numbers: (x^97+19*x+1)*(x^103-19*x^97+14)*(x^100-1)

```
sage.misc.benchmark.bench1()
```

Run a benchmark.

BENCHMARK:

```
sage: from sage.misc.benchmark import *
sage: print(bench1()[0])
```

Find the Mordell-Weil group of the elliptic curve 5077A using mwr ank
Utilities, Release 10.4

```python
>>> from sage.all import *
>>> from sage.misc.benchmark import *
>>> print(bench1()[Integer(0)])
Find the Mordell-Weil group of the elliptic curve 5077A using mwrack
```

`sage.misc.benchmark.bench2()`

Run a benchmark.

**BENCHMARK:**

```python
sage: from sage.misc.benchmark import *
sage: print(bench2()[0])
Some basic arithmetic with very large Integer numbers: '3^1000001 * 19^100001
```

```python
>>> from sage.all import *
>>> from sage.misc.benchmark import *
>>> print(bench2()[Integer(0)])
Some basic arithmetic with very large Integer numbers: '3^1000001 * 19^100001
```

`sage.misc.benchmark.bench3()`

Run a benchmark.

**BENCHMARK:**

```python
sage: from sage.misc.benchmark import *
sage: print(bench3()[0])
Some basic arithmetic with very large Rational numbers: '(2/3)^100001 * (17/19)^
```

```python
>>> from sage.all import *
>>> from sage.misc.benchmark import *
>>> print(bench3()[Integer(0)])
Some basic arithmetic with very large Rational numbers: '(2/3)^100001 * (17/19)^
```

`sage.misc.benchmark.bench4()`

Run a benchmark.

**BENCHMARK:**

```python
sage: from sage.misc.benchmark import *
sage: print(bench4()[0])
Rational polynomial arithmetic using Sage. Compute (x^29+17*x-5)^200.
```

```python
>>> from sage.all import *
>>> from sage.misc.benchmark import *
>>> print(bench4()[Integer(0)])
Rational polynomial arithmetic using Sage. Compute (x^29+17*x-5)^200.
```

`sage.misc.benchmark.bench5()`

Run a benchmark.

**BENCHMARK:**

```python
sage: from sage.misc.benchmark import *
sage: print(bench5()[0])
```

(continues on next page)
Rational polynomial arithmetic using Sage. Compute \((x^{19} - 18x + 1)^{50}\) one hundred times.

```python
>>> from sage.all import *
>>> from sage.misc.benchmark import *
>>> print(bench5()[Integer(0)])
```

**sage.misc.benchmark.bench6()**

Run a benchmark.

**BENCHMARK:**

```python
sage: from sage.misc.benchmark import *
sage: print(bench6()[0])
Compute the \(p\)-division polynomials of \(y^2 = x^3 + 37x - 997\) for primes \(p < 40\).
```

```python
>>> from sage.all import *
>>> from sage.misc.benchmark import *
>>> print(bench6()[Integer(0)])
```

**sage.misc.benchmark.bench7()**

Run a benchmark.

**BENCHMARK:**

```python
sage: from sage.misc.benchmark import *
sage: print(bench7()[0])
Compute the Mordell-Weil group of \(y^2 = x^3 + 37x - 997\).
```

```python
>>> from sage.all import *
>>> from sage.misc.benchmark import *
>>> print(bench7()[Integer(0)])
```

**sage.misc.benchmark.benchmark\((n=-1)\)**

Run a well-chosen range of Sage commands and record the time it takes for each to run.

**INPUT:**

\(n\) – int (default: -1) the benchmark number; the default of -1 runs all the benchmarks.

**OUTPUT:**

list – summary of timings for each benchmark. int – if \(n == -1\), also return the total time

**EXAMPLES:**

```python
sage: from sage.misc.benchmark import *
sage: _ = benchmark()
Running benchmark 0
Benchmark 0: Factor the following polynomial over the rational numbers: \((x^{97}+19*x+1)*(x^{103}-19*x^{97}+14)*(x^{100}-1)\)
Time: ... seconds
```

(continues on next page)
Running benchmark 1
Find the Mordell–Weil group of the elliptic curve 5077A using mwrank
Time: ... seconds
Running benchmark 2
Some basic arithmetic with very large Integer numbers: '3\times1000001 \times 19^{100001}
Time: ... seconds
Running benchmark 3
Some basic arithmetic with very large Rational numbers: '(2/3)^{100001} \times (17/19)^{-100001}
Time: ... seconds
Running benchmark 4
Rational polynomial arithmetic using Sage. Compute \((x^{29}+17x-5)^{200}.
Time: ... seconds
Running benchmark 5
Rational polynomial arithmetic using Sage. Compute \((x^{19} - 18x + 1)^{50}\) one hundred times.
Time: ... seconds
Running benchmark 6
Compute the p-division polynomials of \(y^2 = x^3 + 37x - 997\) for primes \(p < 40\).
Time: ... seconds
Running benchmark 7
Compute the Mordell–Weil group of \(y^2 = x^3 + 37x - 997\).
Time: ... seconds
Running benchmark 8
>>> from sage.all import *
>>> from sage.misc.benchmark import *
>>> _ = benchmark()
Running benchmark 0
Benchmark 0: Factor the following polynomial over
the rational numbers: \((x^{97}+19x+1)(x^{103}-19x^97+14)(x^{100}-1)
Time: ... seconds
Running benchmark 1
Find the Mordell–Weil group of the elliptic curve 5077A using mwrank
Time: ... seconds
Running benchmark 2
Some basic arithmetic with very large Integer numbers: '3\times1000001 \times 19^{100001}
Time: ... seconds
Running benchmark 3
Some basic arithmetic with very large Rational numbers: '(2/3)^{100001} \times (17/19)^{-100001}
Time: ... seconds
Running benchmark 4
Rational polynomial arithmetic using Sage. Compute \((x^{29}+17x-5)^{200}.
Time: ... seconds
Running benchmark 5
Rational polynomial arithmetic using Sage. Compute \((x^{19} - 18x + 1)^{50}\) one hundred times.
Time: ... seconds
Running benchmark 6
Compute the p-division polynomials of \(y^2 = x^3 + 37x - 997\) for primes \(p < 40\).
Time: ... seconds
Running benchmark 7
Compute the Mordell–Weil group of \(y^2 = x^3 + 37x - 997\).
Time: ... seconds
Running benchmark 8

Chapter 7. Development Tools
7.2.2 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:**

```python
sage: timeit('1+1')  # random output
625 loops, best of 3: 314 ns per loop

>>> from sage.all import *
>>> 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

```python
class sage.misc.sage_timeit.SageTimeitResult
```

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:**

```python
sage: from sage.misc.sage_timeit import SageTimeitResult
sage: SageTimeitResult( (3, 5, int(8), pi, 'ms') )
# needs sage.symbolic
3 loops, best of 5: 3.1415927 ms per loop

>>> from sage.all import *
>>> from sage.misc.sage_timeit import SageTimeitResult
>>> SageTimeitResult( (Integer(3), Integer(5), int(Integer(8)), pi, 'ms') )
# needs sage.symbolic
3 loops, best of 5: 3.1415927 ms per loop
```

```python
sage: units = ['s', 'ms', 'μs', 'ns']
sage: scaling = [1, 1e3, 1e6, 1e9]
sage: number = 7
sage: repeat = 13
sage: precision = int(5)
sage: best = pi / 10 ^ 9
# needs sage.symbolic
sage: order = 3
sage: stats = (number, repeat, precision, best * scaling[order], units[order])
```

(continues on next page)
needs sage.symbolic
sage: SageTimeitResult(stats)                           # ...
needs sage.symbolic
7 loops, best of 13: 3.1416 ns per loop

>>> from sage.all import *
>>> units = [u"s", u"ms", u"μs", u"ns"]
>>> scaling = [Integer(1), RealNumber('1e3'), RealNumber('1e6'), RealNumber('1e9')]
>>> number = Integer(7)
>>> repeat = Integer(13)
>>> precision = int(Integer(5))
>>> best = pi / Integer(10) ** Integer(9)
   # needs sage.symbolic
   ...
   ...
>>> order = Integer(3)
>>> stats = (number, repeat, precision, best * scaling[order], units[order]) # ...
   needs sage.symbolic
>>> SageTimeitResult(stats)                           # ...
   needs sage.symbolic
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>

>>> from sage.all import *
>>> SageTimeitResult( (Integer(1), Integer(2), Integer(3), Integer(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, (default: 0), number of loops.
- repeat – integer, (default: 3), number of repetition.
- 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:
Utilities, Release 10.4

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'

>>> from sage.all import *
>>> from sage.misc.sage_timeit import sage_timeit
>>> sage_timeit('3^100000', globals(), preparse=True, number=Integer(50))  # random output
'50 loops, best of 3: 1.97 ms per loop'
>>> sage_timeit('3^100000', globals(), preparse=False, number=Integer(50))  # random output
'50 loops, best of 3: 67.1 ns per loop'
>>> a = Integer(10)
>>> sage_timeit('a^2', globals(), number=Integer(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

>>> from sage.all import *
>>> timeit('10^2', number=Integer(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  # needs sage.modules
3.129814179492283e-07
Sage: min(s.series)  # random output
2.9258728027343752e-07
Sage: t = stats.TimeSeries(s.series)  # needs numpy sage.modules
Sage: t.scale(10^6).plot_histogram(bins=20, figsize=[12, 6], ymax=2)  # needs numpy sage.modules sage.plot
Graphics object consisting of 20 graphics primitives

>>> from sage.all import *
>>> s = sage_timeit('10^2', globals(), repeat=Integer(1000))
>>> len(s.series)
1000
>>> mean(s.series)  # random output  # needs sage.modules
(continues on next page)
3.1298141479492283e-07
>>> min(s.series)  # random output
2.9258728027343752e-07
>>> t = stats.TimeSeries(s.series)  
→ needs numpy sage.modules
>>> t.scale(Integer(10)**Integer(6)).plot_histogram(bins=Integer(20),  
→ figsize=[Integer(12), Integer(6)], ymax=Integer(2))  # needs numpy...
→ sage.modules sage.plot
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(ra = 2
b=131
factor(a^b-1))
sage: sage_timeit(a = 2 + CR + b=131 + CR + factor(a^b-1),  
→ needs sage.libs.pari
...:
   globals(), number=10)
10 loops, best of 3: ... per loop
```

```
>>> from sage.all import *
>>> from sage.misc.sage_timeit import sage_timeit
>>> from os import linesep as CR
>>> # sage_timeit(ra = 2
b=131
factor(a^b-1))
>>> sage_timeit(a = 2 + CR + b=131 + CR + factor(a^b-1),  
→ needs sage.libs.pari
...:
   globals(), number=Integer(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
```

```
>>> from sage.all import *
>>> timeit("print('Hi')", number=Integer(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
1.4255236816e-06
sage: t = timeit("print('Hi')", seconds=True)
sage: t  # r random output
3.6010742187499999e-07
```

```
>>> from sage.all import *
>>> timeit("print('Hi')", seconds=True)  # random output
1.4255236816e-06
>>> t = timeit("print('Hi')", seconds=True)
>>> t  # r random output
3.6010742187499999e-07
```
7.2.3 The timeit command

This uses the function \texttt{sage\_timeit()}.  

\texttt{class sage.misc.sage\_timeit\_class.SageTimeit}  
\texttt{Bases: object}  

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 \texttt{timeit} module, which avoids a number of common traps for measuring execution times. It is also based on IPython's \texttt{\%timeit} command.  

\textbf{TYPICAL INPUT FORMAT:}  

\texttt{timeit(statement, preparse=None, number=0, repeat=3, precision=3)}  

\textbf{EXAMPLES:}  

\begin{verbatim}  
\texttt{sage: timeit('2^10000')}  
625 loops, best of 3: ... per loop  
\end{verbatim}  

\begin{verbatim}  
>>> \texttt{from sage.all import *}  
>>> \texttt{timeit('2^10000')}  
625 loops, best of 3: ... per loop  
\end{verbatim}  

We illustrate some options:  

\begin{verbatim}  
\texttt{sage: timeit('2+2', precision=2, number=20, repeat=5)}  
20 loops, best of 5: ... per loop  
\end{verbatim}  

\begin{verbatim}  
>>> \texttt{from sage.all import *}  
>>> \texttt{timeit('2+2', \texttt{precision=2}, \texttt{number=20}, \texttt{repeat=5})}  
20 loops, best of 5: ... per loop  
\end{verbatim}  

The preparser is on by default (if it is on), but the preparse option allows us to override it:  

\begin{verbatim}  
\texttt{sage: timeit('2^10000', \texttt{preparse=False}, \texttt{number=50})}  
50 loops, best of 3: ... per loop  
\end{verbatim}  

\begin{verbatim}  
>>> \texttt{from sage.all import *}  
>>> \texttt{timeit('2^10000', \texttt{preparse=False}, \texttt{number=Integer(50)})}  
50 loops, best of 3: ... per loop  
\end{verbatim}  

The input can contain newlines:  

\begin{verbatim}  
\texttt{sage: timeit("a = 2\nb=131\nfactor(a^b-1)"}, \texttt{number=25})  
\texttt{\# needs sage.libs.pari}  
25 loops, best of 3: ... per loop  
\end{verbatim}  

\begin{verbatim}  
>>> \texttt{from sage.all import *}  
>>> \texttt{timeit("a = 2\nb=131\nfactor(a^b-1)"}, \texttt{number=Integer(25)})  
\texttt{\# needs sage.libs.pari}  
25 loops, best of 3: ... per loop  
\end{verbatim}  

\textbf{See also:}  

\texttt{runsnake()}
**Utilities, Release 10.4**

**eval** *(code, globs=None, locals=None, **kwds)*

This eval function is called when doing `%timeit` 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'
```

```python
from sage.all import *

>> 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 do not appear in CPU time.

```sage
timeit('sleep(float(0.5))', number=3)    # long time (5s on sage.math, 2012)
3 loops, best of 3: ... ms per loop
```

```python
from sage.all import *

>>> from sage.all import *

>>> timeit('sleep(float(0.5))', number=Integer(3))    # long time (5s on sage.math, 2012)
3 loops, best of 3: ... ms per loop
```

### 7.2.4 Simple profiling tool

**AUTHORS:**
- David Harvey (August 2006)
- Martin Albrecht

```python
class sage.misc.profiler.Profiler(systems=[], verbose=False)
Bases: object

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 :-)

**EXAMPLES:**

```python
from sage.misc.profiler import Profiler

sage: def f():    # not tested
    ....:    p = Profiler()
```

Calling `p(message)` creates a checkpoint:
Do something time-consuming:

```python
sage: x = factor(15) # not tested
>>> from sage.all import *
```

You can create a checkpoints without a string; `Profiler` will use the source code instead:

```python
sage: y = factor(Integer(25))
sage: z = factor(Integer(35))
```

This will give a nice list of timings between checkpoints:

```python
sage: print(p) # not tested
>>> from sage.all import *
```

Let's try it out:

```python
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()

print_last()
  Prints the last profiler step

7.2.5 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 environment variable if you want to change it.

EXAMPLES:

```python
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 = ...
```

```python
>>> from sage.all import *
>>> from sage.misc.gperftools import Profiler, run_100ms
>>> prof = Profiler()
>>> prof.start()  # optional - gperftools
>>> run_100ms()
>>> 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: 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:
Utilities, Release 10.4

sage: from sage.misc.gperftools import Profiler
sage: Profiler()
Profiler logging to ...

>>> from sage.all import *
>>> from sage.misc.gperftools import 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()  # optional - gperftools
'.../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 (default: True). Whether to return cumulative timings.
- `verbose` - boolean (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

(continues on next page)
Utilities, Release 10.4

PROFILE: interrupts/evictions/bytes = ...
>>> f = tmp_filename(ext='.txt')  # optional - gperftools
>>> prof.save(f, verbose=False)  # optional - gperftools

start()

Start profiling

EXAMPLES:

```python
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:

```python
>>> from sage.misc.gperftools import Profiler, run_100ms
>>> prof = Profiler()
>>> prof.start()  # optional - gperftools
>>> run_100ms()
>>> prof.stop()  # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
```  

top(cumulative=True)

Print text report

OUTPUT:

Nothing. A textual report is printed to stdout.

EXAMPLES:

```python
sage: from sage.misc.gperftools import Profiler
sage: prof = Profiler()
sage: prof.start()  # optional - gperftools
```
sage.misc.gperftools.crun(s, evaluator)

Profile single statement.

- s – string. Sage code to profile.
- evaluator – callable to evaluate.

EXAMPLES:

```python
sage: import sage.misc.gperftools as gperf
sage: ev = lambda ex:eval(ex, globals(), locals())
```

```python
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:

```python
sage: from sage.misc.gperftools import run_100ms
sage: run_100ms()
```

```python
>>> from sage.all import *
```

```python
>>> import sage.misc.gperftools as gperf
```

```python
>>> gperf.crun('gperf.run_100ms()', evaluator=ev) # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
Using local file ...
Using local file ...
```
7.3 Miscellaneous Inspection and Development Tools

7.3.1 Dynamic documentation for instances of classes

The functionality in this module allows to define specific docstrings of *instances* of a class, which are different from the class docstring. A typical use case is given by cached methods: the documentation of a cached method should not be the documentation of the class `CachedMethod`; it should be the documentation of the underlying method.

In order to use this, define a class docstring as usual. Also define a method `def _instancedoc_(self)` which should return the docstring of the instance `self`. Finally, add the decorator `@instancedoc` to the class.

**Warning:** Since the `__doc__` attribute is never inherited, the decorator `@instancedoc` must be added to all subclasses of the class defining `_instancedoc_`. Doing it on the base class is not sufficient.

**EXAMPLES:**

```python
sage: from sage.misc.instancedoc import instancedoc
sage: @instancedoc
....: class X:
....:
....:
....:
....:     "Class docstring"
....:     def _instancedoc_(self):
....:         return "Instance docstring"

sage: X.__doc__
'Class docstring'
sage: X().__doc__
'Instance docstring'
```

For a Cython `cdef class`, a decorator cannot be used. Instead, call `instancedoc()` as a function after defining the class:

```python
sage: cython( #_

"needs sage.misc.cython"
....:   '\',
....:   '\',
....:   from sage.misc.instancedoc import instancedoc
....:   cdef class Y:
....:     "Class docstring"
....:     def _instancedoc_(self):
....:         return "Instance docstring"
....:     instancedoc(Y)
....:   '')[

sage: Y.__doc__
'Instance docstring'
```
(continues on next page)
One can still add a custom __doc__ attribute on a particular instance:

```
sage: obj = X()
sage: obj.__doc__ = "Very special doc"
sage: print(obj.__doc__)
Very special doc
```

This normally does not work on extension types:

```
sage: Y().__doc__ = "Very special doc" # needs sage.misc.cython
Traceback (most recent call last):
...
AttributeError: attribute '__doc__' of 'Y' objects is not writable
```

This is an example involving a metaclass, where the instances are classes. In this case, the __instancedoc__ from the metaclass is only used if the instance of the metaclass (the class) does not have a docstring:

```
>>> from sage.all import *
>>> cython()
"needs sage.misc.cython
...
... from sage.misc.instancedoc import instancedoc
... cdef class Y:
...   "Class docstring"
...   def __instancedoc__(self):
...     return "Instance docstring"
... instancedoc(Y)
...
>>> Y().__doc__
"needs sage.misc.cython
... "File:...
Class docstring"
>>> Y().__doc__
"needs sage.misc.cython
... "File:...
Class docstring"
```
Utilities, Release 10.4

sage: @instancedoc
....: class Meta(type):
....:     "Metaclass doc"
....:     def _instancedoc_(self):
....:         return "Docstring for {}".format(self)
sage: class T(metaclass=Meta):
....:     pass
sage: print(T.__doc__)  
Docstring for <class '__main__.T'>
sage: class U(metaclass=Meta):
....:     "Special doc for U"

sage: print(U.__doc__)  
Special doc for U

>>> from sage.all import *
>>> @instancedoc
... class Meta(type):
...     "Metaclass doc"
...     def _instancedoc_(self):
...         return "Docstring for {}".format(self)
>>> class T(metaclass=Meta):
...     pass
>>> print(T.__doc__)  
Docstring for <class '__main__.T'>
>>> class U(metaclass=Meta):
...     "Special doc for U"
>>> print(U.__doc__)  
Special doc for U

class sage.misc.instancedoc.InstanceDocDescriptor
    Bases: object
    Descriptor for dynamic documentation, to be installed as the __doc__ attribute.

    INPUT:
    - classdoc - (string) class documentation
    - instancedoc - (method) documentation for an instance
    - attr - (string, default __doc__) attribute name to use for custom docstring on the instance.

    EXAMPLES:

sage: from sage.misc.instancedoc import InstanceDocDescriptor
sage: def instancedoc(self):
....:     return "Instance doc"
sage: docattr = InstanceDocDescriptor("Class doc", instancedoc)
sage: class Z():
....:     __doc__ = InstanceDocDescriptor("Class doc", instancedoc)
    Z.__doc__
    'Class doc'
    Z().__doc__
    'Instance doc'

>>> from sage.all import *
>>> from sage.misc.instancedoc import InstanceDocDescriptor
>>> def instancedoc(self):

(continues on next page)
... return "Instance doc"
>>> docattr = InstanceDocDescriptor("Class doc", instancedoc)
>>> class Z():
...    __doc__ = InstanceDocDescriptor("Class doc", instancedoc)
>>> Z.__doc__
'Class doc'
>>> Z().__doc__
'Instance doc'

We can still override the __doc__ attribute of the instance:

```python
sage: obj = Z()
sage: obj.__doc__ = "Custom doc"
sage: obj.__doc__
'Custom doc'
sage: del obj.__doc__
sage: obj.__doc__
'Instance doc'
```

```python
>>> from sage.all import *
>>> from sage.misc.sageinspect import *
>>> from sage.all import *
>>> from sage.misc.sageinspect import *
```

```python
sage.misc.instancedoc.instancedoc(cls)
```

Add support for _instancedoc_ to the class cls.

Typically, this will be used as decorator.

INPUT:

- cls – a new-style class

OUTPUT: cls

**Warning:** instancedoc mutates the given class. So you are not supposed to use it as `newcls = instancedoc(cls)` because that would mutate cls (and `newcls` would be the same object as cls)

### 7.3.2 Inspect Python, Sage, and Cython objects

This module extends parts of Python’s inspect module to Cython objects.

**EXAMPLES:**

```python
sage: from sage.misc.sageinspect import *
```

```python
>>> from sage.all import *
>>> 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)
'# distutils: ...Rational Numbers...'
```

```
>>> from sage.all import *
>>> sage_getfile(sage.rings.rational)
'.../rational.pyx'
>>> sage_getdoc(sage.rings.rational).lstrip()
'Rational Numbers...'
>>> sage_getsource(sage.rings.rational)
'# distutils: ...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()[51:-1]
'Inspect Python, Sage, and Cython objects...'
```

```
>>> from sage.all import *
>>> sage_getfile(sage.misc.sageinspect)
'.../sageinspect.py'
>>> print(sage_getdoc(sage.misc.sageinspect).lstrip()[:Integer(40)])
Inspect Python, Sage, and Cython objects
>>> sage_getsource(sage.misc.sageinspect).lstrip()[Integer(51):-Integer(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...'
```

```
>>> from sage.all import *
>>> sage_getfile(sage.rings.rational.Rational)
'.../rational.pyx'
>>> sage_getdoc(sage.rings.rational.Rational).lstrip()
'A rational number...'
>>> sage_getsource(sage.rings.rational.Rational)
'cdef class Rational...'
```

Python classes:

```
sage: sage_getfile(BlockFinder)
'.../sage/misc/sageinspect.py'
```

(continues on next page)
Test introspection of functions defined in Python and Cython files:

Cython functions:

```python
sage: sage_getdef(sage.rings.rational.make_rational, obj_name='mr')
'mr(s)'
sage: sage_getfile(sage.rings.rational.make_rational)
'./rational.ppx'
sage: sage_getdoc(sage.rings.rational.make_rational).lstrip()
'Make a rational number ...'
sage: sage_getsource(sage.rings.rational.make_rational)
@cython.binding(True)
def make_rational(s):
    ...'
```

Python functions:

```python
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):...'
```

(continues on next page)
Unfortunately, no argspec is extractable from builtins. Hence, we use a generic argspec:

```python
sage: sage_getdef('str.find', 'find')
'find(*args, **kwds)'
```

By Issue #9976 and Issue #14017, introspection also works for interactively defined Cython code, and with rather tricky argument lines:

```python
sage: cython('def foo(unsigned int x=1, a=\'\', b={not (2+1==3):\'bar\'}, *args, **kwds): return')
```

```python
sage: print(sage_getsource(foo))
def foo(unsigned int x=1, a='\'', b={not (2+1==3):'bar'}, *args, **kwds): return
```

**AUTHORS:**

- Originally taken from Fernando Perez's IPython
- William Stein: extensive modifications
- William Stein: in `_sage_getargspec_cython()`, a modified version of `inspect.getargspec` from the Python Standard Library, which was taken from IPython for use in Sage
- Nick Alexander: extensions, testing
- Simon King: some extension for Cython, generalisation of SageArgSpecVisitor
- Simon King: in `sage_getsourcelines()`, if a class has no docstring then let the class definition be found starting from the `__init__` method.
- Simon King: in `sage_getsourcelines()`, get source lines for dynamic classes
- Simon King: in `_sage_getargspec_cython()`, return an `ArgSpec`, fix some bugs
- Simon King (2011-09): added `_sage_getsourcelines_name_with_dot()`
• Simon King (2013-02): in \_sage\_getargspec\_cython(), recognise varargs and default values in cython code, and return an ArgSpec

**class** sage.misc.sageinspect.BlockFinder

Bases: object

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 \((\text{type, token, srow\_scol, erow\_ecol, line})\)

**class** sage.misc.sageinspect.SageArgSpecVisitor

Bases: NodeVisitor

A simple visitor class that walks an abstract-syntext 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:**

```python
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
[1, 2, 3]
[1, 2, 3]

v = visitor.visit(ast.parse("\('a'\):\('e',2,[None,{(\text{False:True}),'pi']}\), 37.0:
\text{temp}"").body[0].value)
[(37.0, 'temp'), ('a', ('e', 2, [None, (\text{False:True}), 'pi'])))

v = ast.parse("jc = ['veni', 'vidi', 'vici']").body[0]; v
['veni', 'vidi', 'vici']
```

```bash
>>> from sage.all import *
>>> import ast, sage.misc.sageinspect as sms
>>> visitor = sms.SageArgSpecVisitor()

attys = [x for x in dir(v) if not x.startswith('_')]

\'_attributes' in attys and \'_fields' in attys and \'_col\_offset' in attys
True
```

```python
sage: visitor.visit(v.targets[0])
'jc'
sage: visitor.visit(v.value)
['veni', 'vidi', 'vici']
```

7.3. Miscellaneous Inspection and Development Tools

645
visit_BinOp(node)
Visit a Python AST `ast.BinOp` node.

**INPUT:**

- node – the node instance to visit

**OUTPUT:** the result that `node` represents

**EXAMPLES:**

```python
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
\[11, 15, 6, 2.3333333333333335, 2, 48\]
```

```python
>>> from sage.all import *
>>> import ast, sage.misc.sageinspect as sms
>>> visitor = sms.SageArgSpecVisitor()

>>> vis = lambda x: visitor.visit(ast.parse(x).body[Integer(0)].value)

>>> [vis(d) for d in [('3+(2*4)', '7|8', '5^3', '7/3', '7/3', '3<<4')]]
#indirect doctest
\[11, 15, 6, 2.3333333333333335, 2, 48\]
```

visit_BoolOp(node)

**INPUT:**

- node – the node instance to visit

**OUTPUT:** the result that `node` represents

**EXAMPLES:**

```python
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()

sage: [vis(d) for d in [True and 1, False or 3 or None, 3 and 4]]
#indirect doctest
\[1, 3, 4\]
```

```python
>>> from sage.all import *
>>> import ast, sage.misc.sageinspect as sms

>>> visitor = sms.SageArgSpecVisitor()

>>> vis = lambda x: visitor.visit(ast.parse(x).body[Integer(0)].value)

>>> [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

**EXAMPLES:**
\begin{verbatim}
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]
\end{verbatim}

```python
>>> from sage.all import *
>>> import ast, sage.misc.sageinspect as sms
>>> visitor = sms.SageArgSpecVisitor()
>>> vis = lambda x: visitor.visit_Compare(ast.parse(x).body[Integer(0)].value)
>>> [vis(d) for d in ['1<2==2!=3', '1==1>2', '1<2>1', '1<3<2<4']]
[True, False, True, False]
```

\textit{visit Dict}(node)

Visit a Python AST \texttt{ast.Dict} node.

\textbf{INPUT:}

\begin{itemize}
  \item node – the node instance to visit
\end{itemize}

\textbf{OUTPUT:} the dictionary the \texttt{node} represents

\textbf{EXAMPLES:}

\begin{verbatim}
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: v = [vis(d) for d in [[]], {'1': 'one', 'two': 2, 'other': 'bother'}]
[[], [(1, 'one'), ('other', 'bother'), ('two', 2)]]
\end{verbatim}

```python
>>> from sage.all import *
>>> import ast, sage.misc.sageinspect as sms
>>> visitor = sms.SageArgSpecVisitor()
>>> vis = lambda x: visitor.visit_Dict(ast.parse(x).body[Integer(0)].value)
>>> v = [vis(d) for d in [[]], {'1': 'one', 'two': 2, 'other': 'bother'}]
[[], [(1, 'one'), ('other', 'bother'), ('two', 2)]]
```

\textit{visit List}(node)

Visit a Python AST \texttt{ast.List} node.

\textbf{INPUT:}

\begin{itemize}
  \item node – the node instance to visit
\end{itemize}

\textbf{OUTPUT:} the list the \texttt{node} represents

\textbf{EXAMPLES:}

\begin{verbatim}
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]]
\end{verbatim}

```python
>>> from sage.all import *
>>> import ast, sage.misc.sageinspect as sms
>>> visitor = sms.SageArgSpecVisitor()
>>> vis = lambda x: visitor.visit_List(ast.parse(x).body[Integer(0)].value)
>>> [vis(l) for l in [[]], ['s', 't', 'u'], [[e]], [[], [pi]]]
[[], [s, t, u], [e], [], [pi]]
```

\textbf{(continues on next page)}
Visitor = sms.SageArgSpecVisitor()

vis = lambda x: visitor.visit_list(ast.parse(x).body[0].value)

vis(l) for l in ['', '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: 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 ['foo', 'bar']

---->

sage: type(vis(n)) for n in ['foo', 'bar']


visit_NameConstant (node)
Visit a Python AST ast.NameConstant node.

This is an optimization added in Python 3.4 for the special cases of True, False, and None.

INPUT:
- node – the node instance to visit

OUTPUT: None, True, False

EXAMPLES:

sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()

---->

sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()

---->

sage: vis = lambda x: visitor.visit_NameConstant(ast.parse(x).body[0].value)

---->

sage: vis(n) for n in ['True', 'False', 'None']

---->

sage: type(vis(n)) for n in ['True', 'False', 'None']


Chapter 7. Development Tools

648
visit_Num(node)

Visit a Python AST `ast.Num` node.

INPUT:

- `node` – the node instance to visit

OUTPUT: the number the `node` represents

EXAMPLES:

```python
sage: import ast, sage.misc.sageinspect as sms
g SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_Num(ast.parse(x).body[0].value)
sage: [vis(n) for n in ['123', '0.0']]
[123, 0.0]
```

Note: On Python 3 negative numbers are parsed first, for some reason, as a UnaryOp node.

visit_Str(node)

Visit a Python AST `ast.Str` node.

INPUT:

- `node` – the node instance to visit

OUTPUT: the string the `node` represents

EXAMPLES:

```python
sage: import ast, sage.misc.sageinspect as sms
g SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_Str(ast.parse(x).body[0].value)
sage: ['abstract', 'syntax', 'tr
ee']
```

```python
sage: import ast, sage.misc.sageinspect as sms
g SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_Str(ast.parse(x).body[Integer(0)].value)

```
visit_Tuple (node)
Visit a Python AST `ast.Tuple` node.

INPUT:

- `node` – the node instance to visit

OUTPUT: the tuple the `node` represents

EXAMPLES:

```python
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()

sage: def vis(x): return 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.UnaryOp` node.

INPUT:

- `node` – the node instance to visit

OUTPUT: the result that `node` represents

EXAMPLES:

```python
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()

sage: def vis(x): return visitor.visit_UnaryOp(ast.parse(x).body[0].value)
sage: [vis(d) for d in [+(3*2), -(3*2)]]
[6, -6]
```

visit_arg (node)
Visit a Python AST `ast.arg` node.

This node type is only on Python 3, where function arguments are more complex than just an identifier (e.g. they may also include annotations).

For now we simply return the argument identifier as a string.

INPUT:

- `node` – the node instance to visit
OUTPUT: the argument name

EXAMPLES:

```python
from sage.all import *
from sage.misc.sageinspect import formatannotation

s = "def f(a, b=2, c={'a': [4, 5.5, False]}, d=(None, True)):
    return"

visitor = sms.SageArgSpecVisitor()
args = ast.parse(s).body[0].args.args

[a, 'b', 'c', 'd']
```

```
>>> from sage.all import *
>>> import ast, sage.misc.sageinspect as sms

>>> s = "def f(a, b=2, c={'a': [4, 5.5, False]}, d=(None, True)):
    return"

>>> visitor = sms.SageArgSpecVisitor()

>>> args = ast.parse(s).body[Integer(0)].args.args

[a, 'b', 'c', 'd']
```

*sage.misc.sageinspect.formatannotation*(annotation, base_module=None)

This is taken from Python 3.7's inspect.py; the only change is to add documentation.

INPUT:

- annotation – annotation for a function
- base_module – *(default: None)*

This is only relevant with Python 3, so the doctests are marked accordingly.

EXAMPLES:

```python
from sage.misc.sageinspect import formatannotation

s = inspect.signature(foo)

a = s.parameters['a'].annotation
b = s.parameters['b'].annotation
```

```
>>> from sage.all import *

>>> from sage.misc.sageinspect import formatannotation

>>> import inspect

>>> def foo(a, *, b:int, **kwargs):
    ...
    pass

>>> s = inspect.signature(foo)

>>> a = s.parameters['a'].annotation
```

(continues on next page)
sage.misc.sageinspect.is_function_or_cython_function(obj)

Check whether something is a function.

This is a variant of inspect.isfunction(): We assume that anything which has a genuine __code__ attribute (not using __getattr__ overrides) is a function. This is meant to support Cython functions.

Think twice before using this function (or any function from the inspect or sage.misc.sageinspect modules). Most uses of inspect.isfunction() in ordinary library code can be replaced by callable().

EXAMPLES:

```python
sage: from sage.misc.sageinspect import is_function_or_cython_function
sage: def f():
...     pass
...     return
sage: is_function_or_cython_function(f)
True
sage: is_function_or_cython_function(lambda x:x)
True
sage: from sage.categories.coercion_methods import _mul_parent
sage: is_function_or_cython_function(_mul_parent)
True
sage: is_function_or_cython_function(Integer.digits)  # unbound method
False
sage: is_function_or_cython_function(Integer(Integer(1)).digits)  # bound method
False
```

sage.misc.sageinspect.isclassinstance(obj)

Check if argument is instance of non built-in class

INPUT:

- obj - object
EXAMPLES:

```python
sage: from sage.misc.sageinspect import isclassinstance
sage: isclassinstance(int)
False
sage: class myclass: pass
sage: isclassinstance(myclass())
True
sage: isclassinstance(myclass)
False
sage: class mymetaclass(type): pass
sage: class myclass2(metaclass=mymetaclass): pass
sage: isclassinstance(myclass2)
False

>>> from sage.all import *
>>> from sage.misc.sageinspect import isclassinstance
>>> isclassinstance(int)
False
>>> class myclass: pass
>>> isclassinstance(myclass())
True
>>> isclassinstance(myclass)
False
>>> class mymetaclass(type): pass
>>> class myclass2(metaclass=mymetaclass): pass
>>> isclassinstance(myclass2)
False
```

```python
sage.misc.sageinspect.sage_formatargspec(args, varargs=None, varkw=None, defaults=None, kwonlyargs=(), kwonlydefaults=None, annotations={}, formatarg=<class 'str'>, formatvarargs=None, formatvarkw=None, formatvalue=None, formatreturns=None, formatannotation=None)
```

Format an argument spec from the values returned by `getfullargspec`.

The first seven arguments are (args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations). The other
five arguments are the corresponding optional formatting functions that are called to turn names and values into
strings. The last argument is an optional function to format the sequence of arguments.

This is taken from Python 3.7's inspect.py, where it is deprecated. The only change, aside from documentation
(this paragraph and the next, plus doctests), is to remove the deprecation warning.

Sage uses this function to format arguments, as obtained by `sage_getargspec()`. Since
`sage_getargspec()` works for Cython functions while Python's inspect module does not, it makes
sense to keep this function for formatting instances of `inspect.FullArgSpec`.

EXAMPLES:

```python
sage: from sage.misc.sageinspect import sage_formatargspec
sage: args = ['a', 'b', 'c']
sage: defaults = [3]
sage: sage_formatargspec(args, defaults=defaults)
'(a, b, c=3)'
```

```python
>>> from sage.all import *
>>> from sage.misc.sageinspect import sage_formatargspec
(continues on next page)
```
sage.misc.sageinspect.sage_getargspec(obj)

Return the names and default values of a function's arguments.

INPUT:

• obj – any callable object

OUTPUT:

A named tuple FullArgSpec is returned, as specified by the Python library function inspect.getfullargspec().

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:

```python
sage: from sage.misc.sageinspect import sage_getargspec
sage: def f(x, y, z=1, t=2, *args, **keywords):
    ....:     pass
sage: sage_getargspec(f)
FullArgSpec(args=['x', 'y', 'z', 't'], varargs='args', varkw='keywords',
            defaults=(1, 2), kwonlyargs=[], kwonlydefaults=None, annotations={})
```

We now run sage_getargspec on some functions from the Sage library:

```python
sage: sage_getargspec(identity_matrix)  #...
FullArgSpec(args=['ring', 'n', 'sparse'], varargs=None, varkw=None,
            defaults=(0, False), kwonlyargs=[], kwonlydefaults=None,
            annotations={})
sage: sage_getargspec(factor)
FullArgSpec(args=['n', 'proof', 'int_', 'algorithm', 'verbose'],
            varargs=None, varkw='kwds', defaults=(None, False, 'pari', 0),
            kwonlyargs=[], kwonlydefaults=None, annotations={})
```

```python
>>> from sage.all import *
>>> from sage.misc.sageinspect import sage_getargspec
>>> def f(x, y, z=Integer(1), t=Integer(2), *args, **keywords):
...     pass
>>> sage_getargspec(f)
FullArgSpec(args=['x', 'y', 'z', 't'], varargs='args', varkw='keywords',
            defaults=(1, 2), kwonlyargs=[], kwonlydefaults=None, annotations={})
```
In the case of a class or a class instance, the \texttt{FullArgSpec} of the \_\_new\_, \_\_init\_ or \_\_call\_ method is returned:

\begin{verbatim}
>>> sage_getargspec(factor)
FullArgSpec(args=['n', 'proof', 'int_', 'algorithm', 'verbose'],
            varargs=None, varkw='kwds', defaults=(None, False, 'pari', 0),
            kwonlyargs=[], kwonlydefaults=None, annotations={})

sage_getargspec(P)
\end{verbatim}

\begin{verbatim}
FullArgSpec(args=['base_ring', 'n', 'names', 'order'],
            varargs=None, varkw=kwds, defaults=(degrevlex,),
            kwonlyargs=[], kwonlydefaults=None, annotations={})
\end{verbatim}

The following tests against various bugs that were fixed in Issue \#9976:

\begin{verbatim}
sage: from sage.rings.polynomial.real_roots import bernstein_polynomial_factory
      # needs sage.modules
sage: sage_getargspec(bernstein_polynomial_factory_ratlist.coeffs_bitsize)
      # needs sage.modules
sage: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
      # needs sage.rings.polynomial.pbori
sage: sage_getargspec(BooleanMonomialMonoid.gen)
sage: I = P*[x,y]
sage: sage_getargspec(I.groebner_basis)
      # needs sage.libs.singular
sage: cython("cpdef int foo(x,y) except -1: return 1")
      # needs sage.misc.cython
sage: sage_getargspec(foo)
\end{verbatim}
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)
FullArgSpec(args=['a', 'b', 'c', 'd'], varargs=None, varkw=None, defaults=(1,), kwonlyargs=[], kwonlydefaults=None, annotations={})
```

```
>>> from sage.all import *
>>> from sage.rings.polynomial.real_roots import bernstein_polynomial_factory_ratlist
    # needs sage.modules
>>> sage_getargspec(bernstein_polynomial_factory_ratlist.coeffs_bitsize)  # needs sage.modules
FullArgSpec(args=['self'], varargs=None, varkw=None, defaults=None, kwonlyargs=[], kwonlydefaults=None, annotations={})
```

```
>>> from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
    # needs sage.rings.polynomial.pbori
>>> sage_getargspec(BooleanMonomialMonoid.gen)  # needs sage.rings.polynomial.pbori
FullArgSpec(args=['self', 'i'], varargs=None, varkw=None, defaults=(0,), kwonlyargs=[], kwonlydefaults=None, annotations={})
```

```
>>> I = P*[x,y]
>>> sage_getargspec(I.groebner_basis)  # needs sage.libs.singular
FullArgSpec(args=['self', 'algorithm', 'deg_bound', 'mult_bound', 'prot'], varargs=['args'], varkw=['kwds'], defaults=('!', None, None, False), kwonlyargs=[], kwonlydefaults=None, annotations={})
```

```
>>> cython("cpdef int foo(x,y) except -1: return 1")  # needs sage.misc.cython
>>> sage_getargspec(foo)  # needs sage.misc.cython
FullArgSpec(args=['x', 'y'], varargs=None, varkw=None, defaults=None, kwonlyargs=[], kwonlydefaults=None, annotations={})
```

```
>>> from sage.misc.sageinspect import sage_getdef
>>> sage_getdef(obj, obj_name='')
```

Return the definition header for any callable object.

INPUT:

- `obj` – function
• obj_name – string (default: ")

obj_name is prepended to the output.

EXAMPLES:

```python
sage: from sage.misc.sageinspect import sage_getdef
sage: sage_getdef(identity_matrix)  #...
→ needs sage.modules
'(ring, n=0, sparse=False)'
sage: sage_getdef(identity_matrix, 'identity_matrix')  #...
→ needs sage.modules
'identity_matrix(ring, n=0, sparse=False)'
```

```python
>>> from sage.all import *
>>> from sage.misc.sageinspect import sage_getdef
>>> sage_getdef(identity_matrix)
original documentation
(continues on next page)
```

Check that Issue #6848 has been fixed:

```python
sage: sage_getdef(RDF.random_element) '(min=-1, max=1)'
```

```python
>>> from sage.all import *
>>> from sage.misc.sageinspect import sage_getdef
>>> sage_getdef(RDF.random_element) '(min=-1, max=1)'
```

If an exception is generated, None is returned instead and the exception is suppressed.

```python
sage.misc.sageinspect.sage_getdoc(obj, obj_name='', embedded=False)
```

Return the docstring associated to obj as a string.

If obj is a Cython object with an embedded position in its docstring, the embedded position is stripped.

The optional boolean argument embedded controls the string formatting. It is False by default.

INPUT:

• obj – a function, module, etc.: something with a docstring.

EXAMPLES:

```python
sage: from sage.misc.sageinspect import sage_getdoc
sage: sage_getdoc(identity_matrix)[87:124] #...
→ needs sage.modules
'Return the n x n identity matrix over'
sage: def f(a,b,c,d=1): return a+b+c+d
...
```

```python
sage: import functools
sage: f1 = functools.partial(f, 1,c=2)
sage: f.__doc__ = "original documentation"
sage: f1.__doc__ = "specialised documentation"
sage: sage_getdoc(f1)
'original documentation\n'
```

(continues on next page)
sage.misc.sageinspect.sage_getdoc_original(obj)

Return the unformatted docstring associated to obj as a string.

If obj is a Cython object with an embedded position or signature in its docstring, the embedded information is stripped. If the stripped docstring is empty, then the stripped docstring of obj.__init__ is returned instead.

Feed the results from this into the function sage.misc.sagedoc.format() for printing to the screen.

INPUT:

• obj – a function, module, etc.: something with a docstring.

EXAMPLES:

sage: from sage.misc.sageinspect import sage_getdoc_original

Here is a class that has its own docstring:

sage: print(sage_getdoc_original(sage.rings.integer.Integer))

The :class:`Integer` class represents arbitrary precision integers. It derives from the :class:`Element` class, so integers can be used as ring elements anywhere in Sage.

...
Utilities, Release 10.4

sage: from sage.categories.category import Category
sage: class A(Category):
    def __init__(self):
        The __init__ docstring
sage: sage_getdoc_original(A)
'The __init__ docstring'

sage: class B(Category):
    pass
sage: sage_getdoc_original(B)
'

>>> from sage.all import *
>>> from sage.categories.category import Category
>>> class A(Category):
... def __init__(self):
...    The __init__ docstring
...    pass
>>> sage_getdoc_original(A)
'The __init__ docstring'

>>> class B(Category):
... pass
>>> sage_getdoc_original(B)
'

Old-style classes are supported:

sage: class OldStyleClass:
    def __init__(self):
        The __init__ docstring
    pass
sage: print(sage_getdoc_original(OldStyleClass))
The __init__ docstring

>>> from sage.all import *
>>> class OldStyleClass:
... pass
>>> print(sage_getdoc_original(OldStyleClass))
The __init__ docstring

When there is no __init__ method, we just get an empty string:

sage: class OldStyleClass:
    pass
sage: sage_getdoc_original(OldStyleClass)
'

>>> from sage.all import *
>>> class OldStyleClass:
... pass
>>> sage_getdoc_original(OldStyleClass)
'

If an instance of a class does not have its own docstring, the docstring of its class results:
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:

```python
sage: from sage.misc.sageinspect import sage_getfile
sage: sage_getfile(sage.rings.rational)
'...sage/rings/rational.pyx'
sage: from sage.algebras.steenrod.steenrod_algebra import Sq
    # needs sage.combinat sage.modules
sage_getfile(Sq)
'...sage/algebras/steenrod/steenrod_algebra.py'
sage: sage_getfile(x)
    # needs sage.symbolic
'...sage/symbolic/expression.pyx'
```

The following tests against some bugs fixed in Issue #9976:

```python
sage: obj = sage.combinat.partition_algebra.SetPartitionsAk
    # needs sage.combinat sage.modules
sage: sage_getfile(obj)
'...sage/combinat/partition_algebra.py'
```

```python
>>> from sage.all import *
>>> obj = sage.combinat.partition_algebra.SetPartitionsAk
    # needs sage.combinat sage.modules
>>> sage_getfile(obj)
'...sage/combinat/partition_algebra.py'
```
And here is another bug, fixed in Issue #11298:

```python
sage: P.<x,y> = QQ[]
sage: sage_getfile(P)  #...
   '...sage/rings/polynomial/multi_polynomial_libsingular...
```

```python
>>> from sage.all import *
>>> P = QQ['x, y']; (x, y,) = P._first_ngens(2)
>>> sage_getfile(P)  #...
   '...sage/rings/polynomial/multi_polynomial_libsingular...
```

A problem fixed in Issue #16309:

```python
sage: cython(
   #...
   '...
   class Bar: pass
   #...
   cdef class Foo: pass
   #...''
)

sage: sage_getfile(Bar)  #...
   '...pyx'

sage: sage_getfile(Foo)  #...
   '...pyx'
```

```python
>>> from sage.all import *

>>> cython(
   #...
   '...
   class Bar: pass
   #...
   cdef class Foo: pass
   #...''
)

>>> sage_getfile(Bar)  #...
   '...pyx'

>>> sage_getfile(Foo)  #...
   '...pyx'
```

By Issue #18249, we return an empty string for Python builtins. In that way, there is no error when the user types, for example, `range`?

```python
sage: sage_getfile(range) ''
```

```python
>>> from sage.all import *

>>> sage_getfile(range) ''
```

```python
sage.misc.sageinspect.sage_getfile_relative(obj)
```

Get the file name associated to `obj` as a string.

This is the same as `sage_getfile()`, but if the source file is part of the `sage.*` namespace, it makes the file name relative so that it starts with `sage/`. 

7.3. Miscellaneous Inspection and Development Tools 661
INPUT:

- `obj` – a Sage object, module, etc.

EXAMPLES:

```python
sage: from sage.misc.sageinspect import sage_getfile_relative
sage: sage_getfile_relative(sage.rings.rational)
'sage/rings/rational.pyx'
sage: from sage.algebras.steenrod.steenrod_algebra import Sq
# needs sage.combinat sage.modules
sage: sage_getfile_relative(Sq)
# needs sage.combinat sage.modules
sage: sage_getfile_relative(x)
# needs sage.symbolic
sage: sage_getfile_relative(range)
```

```python
>>> from sage.all import *
>>> from sage.misc.sageinspect import sage_getfile_relative
>>> sage_getfile_relative(sage.rings.rational)
'sage/rings/rational.pyx'
>>> from sage.algebras.steenrod.steenrod_algebra import Sq
# needs sage.combinat sage.modules
>>> sage_getfile_relative(Sq)
# needs sage.combinat sage.modules
>>> sage_getfile_relative(x)
# needs sage.symbolic
>>> sage_getfile_relative(range)
```

`sage.misc.sageinspect.sage_getsource(obj)`

Return the source code associated to `obj` as a string, or None.

INPUT:

- `obj` – function, etc.

EXAMPLES:

```python
sage: from sage.misc.sageinspect import sage_getsource
sage: sage_getsource(identity_matrix)[19:60]
'identity_matrix(ring, n=0, sparse=False):

```

```python
>>> from sage.all import *
>>> from sage.misc.sageinspect import sage_getsource
>>> sage_getsource(identity_matrix)[Integer(19):Integer(60)]
'identity_matrix(ring, n=0, sparse=False):'
```

(continues on next page)
Utilities, Release 10.4

(continued from previous page)

˓→ # needs sage.modules
'identity_matrix(ring, n=0, sparse=False):'

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: # needs sage.modules
sage: sage_getsourcelines(matrix)[1]
21
sage: sage_getsourcelines(matrix)[0][0]
'def matrix(*args, **kwds):

>>> from sage.all import *
>>> from sage.misc.sageinspect import sage_getsourcelines
>>> # needs sage.modules
>>> sage_getsourcelines(matrix)[Integer(1)]
21
>>> sage_getsourcelines(matrix)[Integer(0)][Integer(0)]
'def matrix(*args, **kwds):

Some classes customize this using a _sage_src_lines_ method, which gives the source lines of a class instance, but not the class itself. We demonstrate this for CachedFunction:

sage: # needs sage.combinat
sage: cachedfib = cached_function(fibonacci)
sage: sage_getsourcelines(cachedfib)[0][0]
'def fibonacci(n, algorithm="pari") -> Integer:

sage: sage_getsourcelines(type(cachedfib))[0][0]
'cdef class CachedFunction():

>>> from sage.all import *
>>> # needs sage.combinat
>>> cachedfib = cached_function(fibonacci)
>>> sage_getsourcelines(cachedfib)[Integer(0)][Integer(0)]
'def fibonacci(n, algorithm="pari") -> Integer:

>>> sage_getsourcelines(type(cachedfib))[Integer(0)][Integer(0)]
'cdef class CachedFunction():

sage.misc.sageinspect.sage_getvariablenamexx (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 \( \text{obj} \) to a variable name, then return that variable name. If several variables point to \( \text{obj} \), return a sorted list of those names. If \text{omit\_underscore\_names} \ is \text{True} \ (the \text{default}) \ then omit names starting with an underscore “_”.

EXAMPLES:

```
sage: # needs sage.modules
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']
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.misc.sageinspect import sage_getvariablename
>>> A = random_matrix(ZZ, Integer(100))
>>> sage_getvariablename(A)
'A'
>>> B = A
>>> 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))  #...
# needs sage.modules
[]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> sage_getvariablename(random_matrix(ZZ, Integer(60))) ...  #...
# needs sage.modules
[]
```

7.3.3 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 \text{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:

```python
# make some object obj
sage: edit(obj)  # not tested
```

Now for more details and customization:

```python
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:

```python
sage: m.set_editor("vi")
```

To illustrate:

```python
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:

```python
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_LOCAL/lib/python…/site-packages’, it replaces this by ‘SAGE_ROOT/src’.

EXAMPLES:
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 which 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.cpython)
('...sage/cpython/__init__.py', 0)
```

The following tests against a bug that was fixed in Issue #11298:

```
sage: edit_module.file_and_line(x)  #...
→ needs sage.symbolic
('...sage/symbolic/expression...pyx', ...)
```

```
sage: import sage.misc.edit_module as edit_module
sage: edit_module.file_and_line(x)  #...
→ needs sage.symbolic
('...sage/symbolic/expression...pyx', ...)
```

```
sage.misc.edit_module.set_edit_template(template_string)
```

Set the 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}"")

EDIT /usr/local/sage/src/sage/__init__.py:1

>>> from sage.all import *
>>> from sage.misc.edit_module import set_edit_template
>>> set_edit_template("echo EDIT ${file}:${line}"")

EDIT /usr/local/sage/src/sage/__init__.py:1

sage.misc.edit_module.set_editor(editor_name, opts="")

Set 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')

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: sorted(template_fields(t))
['one', 'three', 'two']

>>> from sage.all import *
>>> from sage.misc.edit_module import template_fields
>>> from string import Template
>>> t = Template("Template ${one} with ${two} and ${three}"")
>>> sorted(template_fields(t))
['one', 'three', 'two']
### 7.3.4 Class inheritance graphs

The function `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:

```python
sage: from sage.rings.polynomial.padics import polynomial_padic_capped_relative_, dense, polynomial_padic_flat
sage: G = class_graph(sage.rings.polynomial.padics); G
Digraph on 6 vertices
sage: G.vertices(sort=True)
['Polynomial', 'Polynomial_generic_cdv', 'Polynomial_generic_dense', 'Polynomial_padic', 'Polynomial_padic_capped_relative_dense', 'Polynomial_padic_flat']
```
We construct the inheritance graph of a given class:

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

>>> from sage.all import *

```  

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

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

```  

```python
>>> from sage.all import *

```  

```python
>>> class_graph([Integer(1), Integer(2), Integer(3)]).edges(sort=True, 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`:

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

```  

```python
>>> from sage.all import *

```  

```python
>>> class_graph(sage.rings.polynomial.padics, depth=Integer(2), as_graph=False)
{Polynomial_padic: ['Polynomial'],
 'Polynomial_padic_capped_relative_dense': ['Polynomial_generic_cdv', '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

### 7.3.5 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 dictionary 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 dictionary has several keys except when \texttt{obj} is an instance of a class.

**EXAMPLES:**

```python
sage: from sage.misc.dev_tools import find_object_modules
sage: find_object_modules(RR)  # needs sage.rings.real_mpfr
{'sage.rings.real_mpfr': ['RR']}

sage: find_object_modules(ZZ)  # needs sage.rings.integer_ring
{'sage.rings.integer_ring': ['ZZ']}
```

```python
>>> from sage.all import *
>>> from sage.misc.dev_tools import find_object_modules

>>> find_object_modules(RR)  # needs sage.rings.real_mpfr
{'sage.rings.real_mpfr': ['RR']}

>>> find_object_modules(ZZ)  # needs sage.rings.integer_ring
{'sage.rings.integer_ring': ['ZZ']}
```

**Note:** It might be a good idea to move this function in \texttt{sage.misc.sageinspect}.

\texttt{sage.misc.dev_tools.find_objects_from_name(name, module_name=None, include_lazy_imports=False)}

Return the list of objects from \texttt{module_name} whose name is \texttt{name}.

**INPUT:**

- \texttt{name} – string
- \texttt{module_name} – string or None:
  - If \texttt{module_name} is None, the function runs through all loaded modules and returns the list of objects whose name matches \texttt{name}.
  - If \texttt{module_name} is a string, then search only in submodules of \texttt{module_name}.
- \texttt{include_lazy_imports} – boolean (default: False); whether to include unresolved lazy imports (i.e., \texttt{LazyImport} objects) in the output.

In order to search through more modules you might use the function \texttt{load_submodules()}.

**EXAMPLES:**

```python
sage: import sage.misc.dev_tools as dt
sage: dt.find_objects_from_name('FareySymbol')  # needs sage.modular
<class 'sage.modular.arithgroup.farey_symbol.Farey'>

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

sage: dt.find_objects_from_name('RR', 'sage')  # needs sage
[Real Field with 53 bits of precision]

sage: dt.find_objects_from_name('RR', 'sympy')  # needs sympy
[RR]
```
Examples that do not belong to the global namespace but in a loaded module:

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

When include
lazy
imports=True is used, several LazyImport objects that are resolving to the same
object may be included in the output:

```python
sage: dt.find_objects_from_name('RR', include_lazy_imports=True)  #...
→needs sage.rings.real_mpfr
[Real Field with 53 bits of precision,
...
Real Field with 53 bits of precision,
RR]
```

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

---

sage.misc.dev_tools.import_statement_string(module, names, lazy)
Utilities, Release 10.4

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')"
```

```
>>> from sage.all import *
>>> import sage.misc.dev_tools as dt
>>> modname = 'sage.misc.dev_tools'
>>> names_and_aliases = [('import_statement_string', 'iss')]
>>> dt.import_statement_string(modname, names_and_aliases, False)
'from sage.misc.dev_tools import import_statement_string as iss'
>>> dt.import_statement_string(modname, names_and_aliases, True)
"lazy_import('sage.misc.dev_tools', 'import_statement_string', 'iss')"
>>> 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'
>>> 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 comma-separated strings of 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) "needs sage.libs.gap"
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:

```
from sage.all import *
from sage.combinat.root_system.weyl_group import WeylGroup
from sage.misc.lazy_attribute import lazy_attribute
```

```
sage: import_statements(WeylGroup, lazy_attribute, lazy=True)  # ...
needs sage.libs.gap
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.arith.misc 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`:

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

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

```
>> from sage.all import *
>>> import_statements(matrix)  #...
→ needs sage.modules
# ** 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
```

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

The strings are allowed to be comma-separated names, and parenthesis are stripped for convenience:

```
sage: import_statements('floor, ceil')  #...
→ needs sage.symbolic
from sage.functions.other import floor, ceil
```

```
>> from sage.all import *
>>> import_statements('floor, ceil')  #...
→ needs sage.symbolic
from sage.functions.other import floor, ceil
```

Specifying a string is also useful for objects that are not imported in the Sage interpreter namespace by default.
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
```

```python
>>> from sage.all import *
>>> 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 detect it:

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

```
sage: import_statements('power')
from sage.arith.power import generic_power as power
```

```python
>>> from sage.all import *
>>> import_statements('FareySymbol')
from sage.modular.arithgroup.farey_symbol import Farey as FareySymbol
>>> import_statements('power')
from sage.arith.power 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('HeckeMonoid')
Traceback (most recent call last):
  ...
LookupError: no object named 'HeckeMonoid'
```

```
sage: from sage.misc.dev_tools import load_submodules
sage: load_submodules(sage.monoids)
sage: import_statements('HeckeMonoid')
from sage.monoids.hecke_monoid import HeckeMonoid
```

```python
>>> from sage.all import *
>>> import_statements('HeckeMonoid')
>>> load_submodules(sage.monoids)
```
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 (Issue #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 Issue #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 Issue #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 Issue #15351:

```python
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.cif import CIF
sage: import_statements(NaN)  # needs sage.symbolic
from sage.symbolic.constants import NaN
sage: import_statements(pi)  # needs sage.symbolic
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
```

Check that a name from the global namespace is properly found (see Issue #23779):

```python
sage: import_statements('log')
from sage.misc.functional import log
```

```python
>>> from sage.all import *
```

```python
>>> import_statements('log')
from sage.misc.functional import log
```

**Note:** The programmers try to made this function as smart as possible. Nevertheless it is far from being perfect.
Utilities, Release 10.4

(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 module.

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
```

```
>>> from sage.all import *
>>> 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)
```

```
>>> from sage.all import *
>>> 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.cone... succeeded
load sage.geometry.cone_catalog... succeeded
load sage.geometry.fan_isomorphism... 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
```

```
>>> from sage.all import *
>>> sage.misc.dev_tools.load_submodules(sage.geometry, "database$|lattice")
load sage.geometry.cone... succeeded
load sage.geometry.cone_catalog... succeeded
load sage.geometry.fan_isomorphism... succeeded
...
load sage.geometry.riemannian_manifolds.surface3d_generators... succeeded
```

(continues on next page)
sage.misc.dev_tools.runsnake(command)

Graphical profiling with runsnake

INPUT:

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

EXAMPLES:

```python
sage: from sage.misc.dev_tools import runsnake
sage: runsnake("list(SymmetricGroup(3))")  # optional - runsnake
```

```python
>>> from sage.all import *
>>> from sage.misc.dev_tools import runsnake
>>> runsnake("list(SymmetricGroup(3))")  # optional - runsnake
```

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

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

```python
>>> from sage.all import *
>>> 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
7.3.6 Function Mangling

This module provides utilities for extracting information about python functions.

AUTHORS:

- Tom Boothby (2009): Original version in Python

```python
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,

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

the following calls are equivalent,

```python
>>> from sage.all import *
>>> def f(x = Integer(10)):
    ...     return min(Integer(1),x)
```

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

```python
sage: def wrap(g):
    ....:     def _g(*args,**kwargs):
    ....:         print("{} {}")
    ....:         return g(*args, **kwargs)
    ....:     return _g
sage: h = wrap(f)
```

(continues on next page)
... def _g(*args,**kwargs):
...     print("/ /".format(args, kwargs))
...     return g(*args, **kwargs)
... return _g
>>> h = wrap(f)
>>> t = h()
() {}
>>> t = h(Integer(10))
(10,) {}
>>> t = h(x=Integer(10))
() {'x': 10}

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

INPUT:

- 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:

```python
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,), ())
```

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

```python
sage: class one:
...     def __init__(self, x = 1):
...         self.x = x
```

(continues on next page)
Utilities, Release 10.4

sage: af = ArgumentFixer(one.__init__, 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)))

>>> from sage.all import *
>>> class one:
...     def __init__(self, x = Integer(1)):
...         self.x = x
...     
>>> af = ArgumentFixer(one.__init__, classmethod=True)
>>> af.fix_to_pos(Integer(1),Integer(2),Integer(3),a=Integer(31),b=Integer(2),
˓
→n=Integer(3))
((1, 2, 3), (('a', 31), ('b', 2), ('n', 3)))

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

\[(e_1, e_2, \ldots, e_k), (n_1, v_1), \ldots, (n_m, v_m)\]

where \(n_1, \ldots, n_m\) are the names of the arguments and \(v_1, \ldots, v_m\) are the values passed in; and \(e_1, \ldots, e_k\) 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 \(n_1, \ldots, n_t\) are in order of the function definition, where \(t\) is the number of named arguments. The remaining names, \(n_{t+1}, \ldots, n_m\) are given in alphabetical order. This is useful to extract the names of arguments, but does not maintain equivalence of

\[
A, K = self.fix_to_pos(...)
sself.f(*A,**dict(K))
\]

and

\[
sself.f(...)
\]

in all cases.

EXAMPLES:

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))))

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

\[(e_1, e_2, \ldots, e_k), ((n_1, v_1), \ldots, (n_m, v_m))\]

where \(n_1, \ldots, n_m\) are the names of the arguments and \(v_1, \ldots, v_m\) are the values passed in; and \(e_1, \ldots, e_k\) 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 \(n_1, \ldots, n_m\) are given in alphabetical order.

EXAMPLES:

sage: from sage.misc.function_mangling import ArgumentFixer
sage: def do_something(a, b, c=Integer(3), *args, **kwargs):
...     print("{} {} {} {} ".format(a, b, c, args,
...          sorted(kwargs.items())))
...     return a + b + c
sage: AF = ArgumentFixer(do_something)
sage: A, K = AF.fix_to_pos(1, 2, 3, 4, 5, 6, f=14, e=16)
sage: print("{} ".format(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)]

>>> from sage.all import *
>>> from sage.misc.function_mangling import ArgumentFixer
>>> def do_something(a, b, c=Integer(3), *args, **kwargs):
...     print("{} {} {} {} ".format(a, b, c, args,
...          sorted(kwargs.items())))
...     return a + b + c
...     return a + b + c
>>> AF = ArgumentFixer(do_something)
7.3.7 ReST index of functions

This module contains a function that generates a ReST index table of functions for use in doc-strings.

```
sage.misc.rest_index_of_methods.doc_index(name)

Attribute an index name to a function.

This decorator can be applied to a function/method in order to specify in which index it must appear, in the index
generated by `gen_thematic_rest_table_index()`.

INPUT:

* name – a string, which will become the title of the index in which this function/method will appear.

EXAMPLES:

```
sage: from sage.misc.rest_index_of_methods import doc_index
sage: @doc_index("Wouhouuuuu")
....: def a():
....:     print("Hey")
sage: a.doc_index
'Wouhouuuuu'
```

```
>>> from sage.all import *
>>> from sage.misc.rest_index_of_methods import doc_index
>>> @doc_index("Wouhouuuuu")
... def a():
...     print("Hey")
>>> a.doc_index
'Wouhouuuuu'
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
INPUT:

- **obj** – a list of functions, a module or a class. If given a list of functions, the generated table will consist of these. If given a module or a class, all functions/methods it defines will be listed, except deprecated or those starting with an underscore. In the case of a class, note that inherited methods are not displayed.

- **names** – a dictionary associating a name to a function. Takes precedence over the automatically computed name for the functions. Only used when list_of_entries is a list.

- **sort** – boolean (default: True); whether to sort the list of methods lexicographically.

- **only_local_functions** – boolean (default: True); if list_of_entries is a module, only_local_functions = True means that imported functions will be filtered out. This can be useful to disable for making indexes of e.g. catalog modules such as `sage.coding.codes_catalog`.

- **root** – module or class (default: None); the module, or class, whose elements are to be listed. This is needed to recover the class when this method is called from `gen_thematic_rest_table_index()` (see Issue #36178).

**Warning:** The ReST tables returned by this function use ‘@’ as a delimiter for cells. This can cause trouble if the first sentence in the documentation of a function contains the ‘@’ character.

EXAMPLES:

```
sage: from sage.misc.rest_index_of_methods import gen_rest_table_index
genie_rest_table_index
sage: print(gen_rest_table_index([graphs.PetersenGraph]))
#.. needs sage.graphs
.. csv-table::
   :class: contentstable
   :widths: 30, 70
   :delim: @
   :func:`sage.graphs.generators.smallgraphs.PetersenGraph` @ Return the
   Petersen Graph.
```

```
>>> from sage.all import *
>>> from sage.misc.rest_index_of_methods import gen_rest_table_index
>>> print(gen_rest_table_index([graphs.PetersenGraph]))
#.. needs sage.graphs
.. csv-table::
   :class: contentstable
   :widths: 30, 70
   :delim: @
   :func:`sage.graphs.generators.smallgraphs.PetersenGraph` @ Return the
   Petersen Graph.
```

The table of a module:

```
sage: print(gen_rest_table_index(sage.misc.rest_index_of_methods))
.. csv-table::
   :class: contentstable
   :widths: 30, 70
   :delim: @
   :func:`sage.misc.rest_index_of_methods.doc_index` @ Attribute an index name...
```

(continues on next page)
.. function:: sage.misc.rest_index_of_methods.gen_rest_table_index

   Return a ReST table describing a list of functions.

.. function:: sage.misc.rest_index_of_methods.gen_thematic_rest_table_index

   Return a ReST string of thematically sorted functions (or methods) of a module
   (or class).

.. function:: sage.misc.rest_index_of_methods.list_of_subfunctions

   Return the functions (resp. methods) of a given module (resp. class) with their names.

>>> from sage.all import *
>>> print(gen_rest_table_index(sage.misc.rest_index_of_methods))

.. csv-table::
   :class: contentstable
   :widths: 30, 70
   :delim: @

   function | description
   -------- |-------------

The table of a class:

```python
sage: print(gen_rest_table_index(Graph)) # needs sage.graphs
```

```python
.. csv-table::
   :class: contentstable
   :widths: 30, 70
   :delim: @

   method | description
   ------ |-------------
```

```python
>>> from sage.all import *
>>> print(gen_rest_table_index(Graph)) # needs sage.graphs
```

```python
.. csv-table::
   :class: contentstable
   :widths: 30, 70
   :delim: @

   method | description
   ------ |-------------
```

sage.misc.rest_index_of_methods.gen_thematic_rest_table_index(root, additional_categories=None, only_local_functions=True)
Return a ReST string of thematically sorted functions (or methods) of a module (or class).

INPUT:

- `root` – the module, or class, whose elements are to be listed.
- `additional_categories` – dictionary (default: None); a dictionary associating a category (given as a string) to a function’s name. Can be used when the decorator `doc_index()` does not work on a function.
- `only_local_functions` – boolean (default: True); if `root` is a module, `only_local_functions = True` means that imported functions will be filtered out. This can be useful to disable for making indexes of e.g. catalog modules such as `sage.coding.codes_catalog`.

EXAMPLES:

```python
sage: from sage.misc.rest_index_of_methods import gen_thematic_rest_table_index,...
       list_of_subfunctions
sage: l = list_of_subfunctions(Graph)[0]  #...
       needs sage.graphs
sage: Graph.bipartite_color in l  #...
       needs sage.graphs
True

>>> from sage.all import *
>>> from sage.misc.rest_index_of_methods import list_of_subfunctions
>>> l = list_of_subfunctions(Graph)[Integer(0)]  # needs sage.graphs
>>> Graph.bipartite_color in l  # needs sage.graphs
True
```

`sage.misc.rest_index_of_methods.list_of_subfunctions(root, only_local_functions=True)`

Return the functions (resp. methods) of a given module (resp. class) with their names.

INPUT:

- `root` – the module, or class, whose elements are to be listed.
- `only_local_functions` – boolean (default: True); if `root` is a module, `only_local_functions = True` means that imported functions will be filtered out. This can be useful to disable for making indexes of e.g. catalog modules such as `sage.coding.codes_catalog`.

OUTPUT:

A pair `(list, dict)` where `list` is a list of function/methods and `dict` associates to every function/method the name under which it appears in `root`.

EXAMPLES:

```python
sage: from sage.misc.rest_index_of_methods import list_of_subfunctions
sage: l = list_of_subfunctions(Graph)[0]  #...
       needs sage.graphs
sage: Graph.bipartite_color in l  #...
       needs sage.graphs
True

>>> from sage.all import *
>>> from sage.misc.rest_index_of_methods import list_of_subfunctions
>>> l = list_of_subfunctions(Graph)[Integer(0)]  # needs sage.graphs
>>> Graph.bipartite_color in l  # needs sage.graphs
True
```
" # needs sage.graphs

>>> Graph.bipartite_color in l
" # needs sage.graphs

True
LOW-LEVEL UTILITIES

8.1 Low-level memory allocation functions

AUTHORS:

• Jeroen Demeyer (2011-01-13): initial version (Issue #10258)
• Jeroen Demeyer (2014-12-14): add more functions (Issue #10257)
• Jeroen Demeyer (2015-03-02): move from c_lib to Cython (Issue #17881)

\[\text{sage.ext.memory}.\text{init} \_\text{memory} \_\text{functions}()\]

Set the MPIR/GMP memory functions to the above functions.

EXAMPLES:

```
sage: from sage.ext.memory import init_memory_functions
sage: init_memory_functions()
```

```
>>> from sage.all import *
>>> from sage.ext.memory import init_memory_functions
>>> init_memory_functions()
```

8.2 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.

\[\text{sage.misc.c3}.\text{C3} \_\text{algorithm}(\text{start}, \text{bases}, \text{attribute}, \text{proper})\]

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 Issue #11943 this implementation was used to compute the list of super categories of a category; see \text{all} \_\text{super} \_\text{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 Issue #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:

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

```
So far so good. However:

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

>>> from sage.all import *

```
sage: Foo._all_bases
Traceback (most recent call last):
...
ValueError: Cannot 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 $\text{Foo}$, since $\text{A}$ and $\text{B}$ impose incompatible orders. If the above was a hierarchy of classes, Python would complain that it cannot calculate a consistent Method Resolution Order.

### 8.3 The C3 algorithm, under control of a total order

#### 8.3.1 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 :-(
8.3.2 The problem

Consider the following hierarchy of classes:

```python
sage: class A1():
    pass
sage: class A2():
    ....:    def foo(self): return 2
sage: class A3():
    pass
sage: class A4():
    ....:    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):

```python
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:

```python
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`):
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:

```python
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']
```

```python
>>> from sage.all import *
>>> class A7(A6, A5, A4, A3, A2, A1): pass
>>> [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:

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

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

### 8.3.3 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 On the order of super categories).

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.

### 8.3.4 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 superclasses 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 hierarchy 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:

```python
sage: P = Poset({10: [9,8,7], 9: [6,1], 8: [5,2], 7: [4,3],
               # needs sage.graphs
               6: [3,2], 5: [3,1], 4: [2,1]},
              linear_extension=True, facade=True)
```

694 Chapter 8. Low-Level Utilities
Utilities, Release 10.4

>>> from sage.all import *
>>> P = Poset({Integer(10): [Integer(9), Integer(8), Integer(7)],
              Integer(9): [Integer(6), Integer(1)],
              Integer(8): [Integer(5), Integer(2)],
              Integer(7): [Integer(4), Integer(3)],
              Integer(6): [Integer(3), Integer(2)],
              Integer(5): [Integer(3),
              Integer(1)],
              Integer(4): [Integer(2), Integer(1)]},
              linear_extension=True, facade=True)

And build a HierarchyElement from it:

sage: from sage.misc.c3_controlled import HierarchyElement
sage: x = HierarchyElement(10, P) # needs sage.graphs

Here are its bases:

sage: HierarchyElement(10, P)._bases # needs sage.graphs
[9, 8, 7]

Using the standard C3 algorithm fails:

sage: x.mro_standard # needs sage.graphs
Traceback (most recent call last):
  ...
ValueError: Cannot 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) # needs sage.graphs
sage: list(P) # needs sage.graphs
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

8.3. The C3 algorithm, under control of a total order 695
Now we play with a specific linear extension of $P$:

```python
# needs sage.graphs
Q = P.linear_extension([10, 9, 8, 7, 6, 5, 4, 1, 2, 3]).to_poset()
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], [5, 1], [4, 3], [4, 2]]
```

```python
x = HierarchyElement(10, Q)
x.mro_standard
```

```
Traceback (most recent call last):
...
ValueError: Cannot merge the items 2, 3, 3.
```

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

```python
x.mro
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```python
x.mro_controlled
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

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

```python
x._test_mro()
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```
In practice, the control was achieved by adding the following bases:

```python
sage: x._bases
# needs sage.graphs
[9, 8, 7]
sage: x._bases_controlled
# needs sage.graphs
[9, 8, 7, 6, 5]
```

Altogether, four bases were added for control:

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

This information can also be recovered with:

```python
sage: x.all_bases_len()
# needs sage.graphs
15
sage: x.all_bases_controlled_len()
# needs sage.graphs
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:

8.3. The C3 algorithm, under control of a total order
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:

```python
>>> from sage.all import *
>>> L = P.linear_extensions()  # ...
>>> stats = []                 # ...
>>> 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())
```

We now consider a hierarchy of categories:

```python
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
```

(continues on next page)
For a typical category, few bases, if any, need to be added to force C3 to give the desired order:

```python
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()
72
sage: x.all_bases_controlled_len()
76

sage: C = GradedHopfAlgebrasWithBasis(QQ)
sage: x = HierarchyElement(C, attrcall("super_categories"), attrgetter("_cmp_key"))
sage: x._test_mro()
False
sage: x.all_bases_len()
114
sage: x.all_bases_controlled_len()
117
```

```python
>>> from sage.all import *
>>> C = FiniteFields()
>>> x = HierarchyElement(C, attrcall("super_categories"), attrgetter("_cmp_key"))
>>>
```
The following can be used to search through the Sage named categories for any that requires the addition of some bases. The output may change a bit when the category hierarchy is changed. As long as the list below does not change radically, it’s fine to just update this doctest:

```python
sage: from sage.categories.category import category_sample
sage: sorted((C for C in category_sample())
˓→needs sage.combinat sage.graphs sage.modules sage.rings.number_field
˓→if len(C._super_categories_for_classes) != len(C.super_categories()),
˓→key=str)
[Category of affine Weyl groups,
 Category of fields,
 Category of finite Weyl groups,
 Category of finite dimensional Hopf algebras with basis over Rational Field,
 Category of finite dimensional algebras with basis over Rational Field,
 Category of finite enumerated permutation groups,
 Category of number fields]
```

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:

```python
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:

```python
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:

```python
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]`:

```python
sage: C3_sorted_merge([[1],[2]])
([2, 1], [2, 1])
```
And indeed \texttt{C3\_merge()} now returns the desired result:

\begin{verbatim}
sage: C3_merge([[1],[2,1]])
[2, 1]
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> C3_merge([[Integer(1)],[Integer(2),Integer(1)]]
[2, 1]
\end{verbatim}

From now on, we use this little wrapper that checks that \texttt{C3\_merge()}, with the suggestion of \texttt{C3\_sorted\_merge()}, returns a sorted list:

\begin{verbatim}
sage: def C3_sorted_merge_check(lists):
...    result, suggestion = C3_sorted_merge(lists)
...    assert result == C3_merge(lists[:-1] + [suggestion])
...    return result, suggestion
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> def C3_sorted_merge_check(lists):
...    result, suggestion = C3_sorted_merge(lists)
...    assert result == C3_merge(lists[:-Integer(1)] + [suggestion])
...    return result, suggestion
\end{verbatim}

Base cases:

\begin{verbatim}
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([[1],[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])
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> C3_sorted_merge_check([])
Traceback (most recent call last):
...
ValueError: The input should be a non empty list of lists (or iterables)
>>> C3_sorted_merge_check([[]])
([], [])
>>> C3_sorted_merge_check([[Integer(1)]]
([], [])
>>> C3_sorted_merge_check([[Integer(1)],[1]])
([], [1])
>>> C3_sorted_merge_check([[Integer(3),Integer(2),Integer(1)]]
([], [1])
>>> C3_sorted_merge_check([[Integer(1)],[Integer(1)]]
([], [1])
>>> C3_sorted_merge_check([[Integer(3),Integer(2),Integer(1)],[Integer(3), Integer(2),Integer(1)]]
([], [1])
\end{verbatim}

Exercise different states for the last line:
```python
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:

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

```python
>>> from sage.all import *
>>> C3_sorted_merge_check([[Integer(3),Integer(1)],[Integer(4),Integer(2)]]
([4, 3, 2, 1], [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 Issue #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)

704 Chapter 8. Low-Level Utilities
Todo:

- Handle nicely (covariant) functorial constructions and axioms

EXAMPLES:

```python
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, ...)
```

```python
>>> from sage.all import *
```

```python
>>> Objects()._cmp_key
(0, 0)
>>> SetsWithPartialMaps()._cmp_key
(0, 1)
>>> Sets()._cmp_key
(0, 2)
>>> Sets().Facade()._cmp_key
(1, ...)
```

(continues on next page)
For backward compatibility we currently want the following comparisons:

```python
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
```

```python
>>> from sage.all import *

>>> EnumeratedSets()._cmp_key > Sets().Facade()._cmp_key
True

>>> AdditiveMagmas()._cmp_key > EnumeratedSets()._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

>>> from sage.all import *

... Algebras(GF(Integer(3)))._cmp_key == Algebras(GF(Integer(5)))._cmp_key  #
... indirect doctest
True
>>> Algebras(ZZ)._cmp_key != Algebras(GF(Integer(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 instances of HierarchyElement, and MROs are given a list of the corresponding values.

8.3. The C3 algorithm, under control of a total order

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, originally taken from the hierarchy of categories above **HopfAlgebrasWithBasis**:

```
sage: from sage.misc.c3_controlled import HierarchyElement
sage: G = DiGraph({
--------------
˓→ needs sage.graphs
....:
44 : [43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, ...
˓→ 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7,
˓→ 6, 5, 4, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
43 : [42, 41, 40, 36, 35, 39, 38, 37, 33, 32, 31, 30, 29, 28, 27, 26, ...
˓→ 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, ...
˓→ 1, 0],
˓→ needs sage.graphs
....:
42 : [36, 35, 37, 30, 29, 28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, ...
˓→ 0],
˓→ needs sage.graphs
....:
41 : [40, 36, 35, 33, 32, 31, 30, 29, 28, 27, 26, 23, 22, 21, 20, 19, ...
˓→ 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
40 : [36, 35, 32, 31, 30, 29, 28, 27, 26, 19, 18, 17, 16, 15, 14, 13, ...
˓→ 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
39 : [38, 37, 33, 32, 31, 30, 29, 28, 27, 26, 23, 22, 21, 20, 19, ...
˓→ 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
38 : [37, 33, 32, 31, 30, 29, 28, 27, 26, 23, 22, 21, 20, 19, 18, ...
˓→ 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
37 : [30, 29, 28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
36 : [35, 30, 29, 28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
35 : [29, 28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
34 : [33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, ...
˓→ 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
33 : [32, 31, 30, 29, 28, 27, 26, 23, 22, 21, 20, 19, 18, 17, 16, ...
˓→ 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
32 : [31, 30, 29, 28, 27, 26, 19, 18, 17, 16, 15, 14, 13, 12, 11, ...
˓→ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
31 : [30, 29, 28, 27, 26, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, ...
˓→ 3, 2, 1, 0],
˓→ needs sage.graphs
....:
30 : [29, 28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
29 : [28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
28 : [27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
27 : [15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
26 : [15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
25 : [24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, ...
˓→ 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
24 : [4, 2, 1, 0],
˓→ needs sage.graphs
....:
23 : [22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, ...
˓→ 6, 5, 4, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
22 : [21, 20, 18, 17, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
˓→ needs sage.graphs
....:
21 : [20, 17, 4, 2, 1, 0],
˓→ needs sage.graphs
...
(continues on next page)
Utilities, Release 10.4

(continued from previous page)

....:  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: # needs sage.combinat sage.graphs
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, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
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'>, <class '39.cls'>, <class '38.cls'>, <class '37.cls'>, <class '36.cls'>, <class '35.cls'>, <class '34.cls'>, <class '33.cls'>, <class '32.cls'>, <class '31.cls'>, <class '30.cls'>, <class '29.cls'>, <class '28.cls'>, <class '27.cls'>, <class '26.cls'>, <class '25.cls'>, <class '24.cls'>, <class '23.cls'>, <class '22.cls'>, <class '21.cls'>, <class '20.cls'>, <class '19.cls'>, <class '18.cls'>, <class '17.cls'>, <class '16.cls'>, <class '15.cls'>, <class '14.cls'>, <class '13.cls'>, <class '12.cls'>, <class '11.cls'>, <class '10.cls'>, <class '9.cls'>, <class '8.cls'>, <class '7.cls'>, <class '6.cls'>, <class '5.cls'>, <class '4.cls'>, <class '3.cls'>, <class '2.cls'>, <class '1.cls'>, <... 'object'>]

>>> from sage.all import *
>>> from sage.misc.c3_controlled import HierarchyElement
>>> G = DiGraph({
...   # needs sage.graphs
...   Integer(44) : [Integer(43), Integer(42), Integer(41), Integer(40),
...   Integer(39), Integer(38), Integer(37), Integer(36), Integer(35), Integer(34),
...   Integer(33), Integer(32), Integer(31), Integer(30), Integer(29), Integer(28),
...   Integer(27), Integer(26), Integer(25), Integer(24), Integer(23), Integer(22),
...   Integer(21), Integer(20), Integer(19), Integer(18), Integer(17), Integer(16),
...   Integer(15), Integer(14), Integer(13), Integer(12), Integer(11), Integer(10),
...   Integer(9), Integer(8), Integer(7), Integer(6), Integer(5), Integer(4),
... })

8.3. The C3 algorithm, under control of a total order

(continues on next page)
\[ Integer(3), Integer(2), Integer(1), Integer(0) ],
\[ \ldots \]
\[ Integer(43), Integer(42), Integer(41), Integer(40), Integer(39), \ldots \]
\[ Integer(2), Integer(1), Integer(0) ],
\[ \ldots \]
\[ Integer(0) ],
\[ \ldots \]
\[ Integer(35), Integer(34), Integer(33), Integer(32), \ldots \]
\[ Integer(2), Integer(1), Integer(0) ],
\[ \ldots \]
\[ Integer(0) ],
\[ \ldots \]
\[ Integer(33), Integer(32), Integer(31), Integer(30), \ldots \]
\[ Integer(2), Integer(1), Integer(0) ],
\[ \ldots \]
\[ Integer(0) ],
\[ \ldots \]
\[ Integer(31), Integer(30), Integer(29), Integer(28), \ldots \]
\[ Integer(2), Integer(1), Integer(0) ],
\[ \ldots \]
\[ Integer(0) ],
\[ \ldots \]
\[ Integer(30), Integer(29), Integer(28), \ldots \]
\[ Integer(2), Integer(1), Integer(0) ],
\[ \ldots \]
\[ Integer(0) ],
\[ \ldots \]
\[ Integer(29), Integer(28), Integer(27), \ldots \]
\[ Integer(2), Integer(1), Integer(0) ],
\[ \ldots \]
\[ Integer(0) ],
\[ \ldots \]
8.3. The C3 algorithm, under control of a total order
all_bases()

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

EXAMPLES:

```
sage: # needs sage.graphs
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]

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

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

```python
>>> from sage.all import *
>>> from sage.misc.c3_controlled import HierarchyElement
>>> P = Poset((divisors(Integer(30)), lambda x, y: y.divides(x)), facade=True)  # needs sage.graphs
>>> HierarchyElement(Integer(30), P).all_bases_controlled_len()  # needs sage.graphs
13
```

**all_bases_len()**

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

**EXAMPLES:**

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

```python
>>> from sage.all import *
>>> from sage.misc.c3_controlled import HierarchyElement
>>> P = Poset((divisors(Integer(30)), lambda x, y: y.divides(x)), facade=True)  # needs sage.graphs
>>> HierarchyElement(Integer(30), P).all_bases_len()  # needs sage.graphs
12
```

8.3. The C3 algorithm, under control of a total order
**bases()**

The bases of `self`.

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

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.graphs
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]
```

```python
>>> from sage.all import *

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

**cls()**

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

**EXAMPLES:**

```python
sage: # needs sage.graphs
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'>, <... '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'>, <class '3.cls'>, <class '2.cls'>, <class '1.cls'>, <...]
```

```python
>>> from sage.all import *

>>> # needs sage.graphs
```

(continues on next page)
>>> from sage.misc.c3_controlled import HierarchyElement
>>> P = Poset((divisors(Integer(30)), lambda x, y: y.divides(x)), facade=True)
>>> x = HierarchyElement(Integer(1), P)
>>> x.cls
<class '1.cls'>
>>> x.cls.mro()
[<class '1.cls'>, <... 'object'>]
>>> x = HierarchyElement(Integer(30), P)
>>> x.cls
<class '30.cls'>
>>> x.cls.mro()
[<class '30.cls'>, <class '15.cls'>, <class '10.cls'>, <class '6.cls'>,
  <class '5.cls'>, <class '3.cls'>, <class '2.cls'>, <class '1.cls'>, <... 
  'object'>]

mro()

The MRO for this object, calculated with \texttt{C3\_sorted\_merge}().

EXAMPLES:

\begin{verbatim}
sage: # needs sage.graphs
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])
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> # needs sage.graphs
>>> from sage.misc.c3_controlled import HierarchyElement, C3_sorted_merge,
    \_identity
>>> P = Poset({Integer(7): [Integer(5), Integer(6)], Integer(5): [Integer(1),
    Integer(2)], Integer(6): [Integer(3), Integer(4)]}, facade=True)
>>> x = HierarchyElement(Integer(5), P)
>>> x.mro
[5, 2, 1]
>>> x = HierarchyElement(Integer(6), P)
>>> x.mro
[6, 4, 3]
>>> x = HierarchyElement(Integer(7), P)
>>> x.mro
[7, 6, 5, 4, 3, 2, 1]
>>> C3_sorted_merge([[Integer(6), Integer(4), Integer(3)], [Integer(5),
    Integer(2), Integer(1)], [Integer(6), Integer(5)]], \_identity)
([6, 5, 4, 3, 2, 1], [6, 5, 4])
\end{verbatim}
mro_controlled()

The MRO for this object, calculated with \texttt{C3\_merge()}, under control of \texttt{C3\_sorted\_merge()}

EXAMPLES:

```python
sage: from sage.misc.c3_controlled import HierarchyElement, C3_merge
def needs_sage_graphs():
    return ([6, 5, 4, 3, 2, 1], [6, 5, 4])

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]
```

```python
>>> from sage.all import *
>>> from sage.misc.c3_controlled import HierarchyElement, C3_merge

>>> needs_sage_graphs()


>>> x = HierarchyElement(Integer(5), P)

>>> x.mro_controlled

[5, 2, 1]

>>> x = HierarchyElement(Integer(6), P)

>>> x.mro_controlled

[6, 4, 3]

>>> x = HierarchyElement(Integer(7), P)

>>> x.mro_controlled

[7, 6, 5, 4, 3, 2, 1]

>>> x._bases

[6, 5]

>>> x._bases_controlled

[6, 5, 4]

>>> C3_merge([[Integer(6), Integer(4), Integer(3)], [Integer(5), Integer(2), Integer(1)]], [Integer(6), Integer(5)])

[6, 4, 3, 5, 2, 1]

>>> C3_merge([[Integer(6), Integer(4), Integer(3)], [Integer(5), Integer(2), Integer(1)]], [Integer(6), Integer(5), Integer(4)])

[6, 5, 4, 3, 2, 1]
```
**mro_standard()**

The MRO for this object, calculated with **C3_merge()**

**EXAMPLES:**

```python
sage: from sage.misc.c3_controlled import HierarchyElement, C3_merge

sage: # needs sage.graphs
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]
```

```python
>>> from sage.all import *
>>> from sage.misc.c3_controlled import HierarchyElement, C3_merge

>>> # needs sage.graphs

>>> P = Poset({7: [5, 6], 5: [1, 2], 6: [3, 4]}, facade=True)

>>> x = HierarchyElement(5, P)

>>> x.mro_standard
[5, 2, 1]

>>> x = HierarchyElement(6, P)

>>> x.mro_standard
[6, 4, 3]

>>> x = HierarchyElement(7, P)

>>> x.mro_standard
[7, 6, 4, 3, 5, 2, 1]

>>> C3_merge([[6, 4, 3], [5, 2, 1], [6, 5]])

[6, 4, 3, 5, 2, 1]
```

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

**EXAMPLES:**

```python
sage: from sage.misc.c3_controlled import identity

sage: identity(10)
10
```

```python
>>> from sage.all import *
>>> from sage.misc.c3_controlled import identity

>>> identity(Integer(10))
10
```

8.3. The C3 algorithm, under control of a total order
INDICES AND TABLES

• Index
• Module Index
• Search Page
sage.arith.srange, 138
sage.databases.sql_db, 158
sage.ext.fast_callable, 322
sage.ext.fast_eval, 355
sage.ext.memory, 689
sage.misc.abstract_method, 91
sage.misc.banner, 569
sage.misc.benchmark, 623
sage.misc.bindable_class, 96
sage.misc.c3, 689
sage.misc.c3_controlled, 691
sage.misc.cachefunc, 262
sage.misc.call, 119
sage.misc.callable_dict, 120
sage.misc.citation, 603
sage.misc.classcall_metaclass, 109
sage.misc.classgraph, 668
sage.misc.constant_function, 108
sage.misc.converting_dict, 121
sage.misc.copying, 605
sage.misc.cython, 357
sage.misc.decorators, 99
sage.misc.defaults, 1
sage.misc.dev_tools, 669
sage.misc.edit_module, 664
sage.misc.element_with_label, 144
sage.misc.explain_pickle, 538
sage.misc.fast_methods, 116
sage.misc.flatten, 125
sage.misc.fpickle, 536
sage.misc.func_persist, 519
sage.misc.function_mangling, 680
sage.misc.functional, 3
sage.misc.gperftools, 634
sage.misc.html, 451
sage.misc.inherit_comparison, 113
sage.misc.inline_fortran, 360
sage.misc.instancedoc, 638
sage.misc.latex, 458
sage.misc.latex_macros, 485
sage.misc.latex_standalone, 487
sage.misc.lazy_attribute, 227
sage.misc.lazy_format, 240
sage.misc.lazy_import, 242
sage.misc.lazy_import_cache, 249
sage.misc.lazy_list, 250
sage.misc.lazy_string, 260
sage.misc.map_threaded, 137
sage.misc.mathml, 457
sage.misc.messaging, 153
sage.misc.method_decorator, 115
sage.misc.misc, 197
sage.misc.misc_c, 213
sage.misc.mrange, 127
sage.misc.multireplace, 136
sage.misc.nested_class, 555
sage.misc.object_multiplexer, 116
sage.misc.package, 595
sage.misc.pager, 579
sage.misc.parser, 362
sage.misc.persist, 519
sage.misc.prandom, 77
sage.misc.profiler, 632
sage.misc.python, 378
sage.misc.random_testing, 619
sage.misc.randstate, 63
sage.misc.remote_file, 152
sage.misc.repr, 407
sage.misc.reset, 572
sage.misc.rest_index_of_methods, 684
sage.misc.sage_eval, 380
sage.misc.sage_input, 410
sage.misc.sage_ostools, 154
sage.misc.sage_timeit, 627
sage.misc.sage_timeit_class, 631
sage.misc.sage_unittest, 607
sage.misc.sagedoc, 579
sage.misc.sagedoc_conf, 594
sage.misc.sageinspect, 641
sage.misc.search, 127
sage.misc.session, 561
sage.misc.sh, 384
sage.misc.sphinxify, 594
sage.misc.stopgap, 191
sage.misc.superseded, 193
sage.misc.table, 441
sage.misc.temporary_file, 145
sage.misc.test_nested_class, 622
sage.misc.unknown, 86
sage.misc.verbose, 224
sage.misc.viewer, 574
sage.misc.weak_dict, 306

t
sage.typeset.ascii_art, 396
sage.typeset.character_art, 389
sage.typeset.character_art_factory, 392
sage.typeset.symbols, 385
sage.typeset.unicode_art, 402
sage.typeset.unicode_characters, 405
INDEX

Non-alphabetical

_ rich_repr_() (sage.misc.html.HtmlFragment method), 453
_ rich_repr_() (sage.misc.table.table method), 448

A

abs() (sage.ext.fast_callable.Expression method), 331
abstract_method() (in module sage.misc.abstract_method), 92
abstract_methods_of_class() (in module sage.misc.abstract_method), 95
AbstractMethod (class in sage.misc.abstract_method), 91
add_column() (sage.databases.sql_db.SQLDatabase method), 163
add_data() (sage.databases.sql_db.SQLDatabase method), 165
add_document_class_option() (sage.misc.latex_standalone.Standalone method), 495
add_library() (sage.misc.inline_fortran.InlineFortran method), 360
add_library_path() (sage.misc.inline_fortran.InlineFortran method), 360
add_macro() (sage.misc.latex_standalone.Standalone method), 495
add_macro() (sage.misc.latex.Latex method), 459
add_package_to_preamble_if_available() (sage.misc.latex.Latex method), 459
add_row() (sage.databases.sql_db.SQLDatabase method), 165
add_rows() (sage.databases.sql_db.SQLDatabase method), 166
add_standalone_config() (sage.misc.latex_standalone.Standalone method), 496
add_to_preamble() (sage.misc.latex.Latex method), 460
add_usepackage() (sage.misc.latex_standalone.Standalone method), 496
addusetikzlibrary() (sage.misc.latex_standalone.TikzPicture method), 508
additive_order() (in module sage.misc.functional), 8

all_bases() (sage.misc.c3_controlled.HierarchyElement method), 712
all_bases_controlled_len() (sage.misc.c3_controlled.HierarchyElement method), 713
all_bases_len() (sage.misc.c3_controlled.HierarchyElement method), 713
APPEND() (sage.misc.explain_pickle.PickleExplainer method), 541
append() (sage.misc.explain_pickle.TestAppendList method), 548
APPENDS() (sage.misc.explain_pickle.PickleExplainer method), 541
apply() (sage.misc.sagedoc_conf.SagemathTransform method), 594
ArgumentFixer (class in sage.misc.function_mangling), 680
arguments() (sage.ext.fast_callable.ExpressionCall method), 332
AsciiArt (class in sage.typeset.ascii_art), 400
assign() (sage.misc.sage_input.SageInputBuilder method), 421
atomic_dir (class in sage.misc.temporary_file), 145
atomic_write (class in sage.misc.temporary_file), 146
attrcall() (in module sage.misc.sage.call), 119
AttrCallObject (class in sage.misc.call), 119
attributes() (in module sage.misc.lazy_import), 243

B

BackslashOperator (class in sage.misc.misc), 197
backtrack() (sage.misc.parser.Tokenizer method), 374
balanced_sum() (in module sage.misc.functional), 214
banner() (in module sage.misc.banner), 569
banner_text() (in module sage.misc.banner), 569
base() (sage.ext.fast_callable.ExpressionIPow method), 335
base_field() (in module sage.misc.functional), 8
base_ring() (in module sage.misc.functional), 9
bases() (sage.misc.c3_controlled.HierarchyElement method), 713
basis() (in module sage.misc.functional), 9
bench0() (in module sage.misc.benchmark), 623
bench1() (in module sage.misc.benchmark), 623
bench2() (in module sage.misc.benchmark), 624
bench3() (in module sage.misc.benchmark), 624
bench4() (in module sage.misc.benchmark), 624
bench5() (in module sage.misc.benchmark), 624
bench6() (in module sage.misc.benchmark), 625
bench7() (in module sage.misc.benchmark), 625
benchmark() (in module sage.misc.benchmark), 625
benchmark_libc() (in module sage.misc.benchmark), 71
benchmark_mt() (in module sage.misc.randstate), 71
BindableClass (class in sage.misc.bindable_class), 96
BINFLOAT() (sage.misc.explain_pickle.PickleExplainer method), 541
BINGET() (sage.misc.explain_pickle.PickleExplainer method), 541
BININT() (sage.misc.explain_pickle.PickleExplainer method), 541
BININT1() (sage.misc.explain_pickle.PickleExplainer method), 541
BININT2() (sage.misc.explain_pickle.PickleExplainer method), 541
BINPERSID() (sage.misc.explain_pickle.PickleExplainer method), 541
BININPUT() (sage.misc.explain_pickle.PickleExplainer method), 541
BINSTRING() (sage.misc.explain_pickle.PickleExplainer method), 541
BINUNICODE() (sage.misc.explain_pickle.PickleExplainer method), 541
blackboard_bold() (sage.misc.latex.Latex method), 460
BlockFinder (class in sage.misc.sageinspect), 645
bool_function() (in module sage.misc.latex), 472
bool_function() (in module sage.misc.mathml), 457
BoundClass (class in sage.misc.bindable_class), 99
browser() (in module sage.misc.viewer), 577
browser() (sage.misc.viewer.Viewer method), 575
BUILD() (sage.misc.explain_pickle.PickleExplainer method), 541
build() (sage.typeset.character_art_factory.CharacterArtFactory method), 392
build_container() (sage.typeset.character_art_factory.CharacterArtFactory method), 393
build_dict() (sage.typeset.character_art_factory.CharacterArtFactory method), 393
build_empty() (sage.typeset.character_art_factory.CharacterArtFactory method), 393
build_from_magic_method() (sage.typeset.character_art_factory.CharacterArtFactory method), 394
build_from_string() (sage.typeset.character_art_factory.CharacterArtFactory method), 394
build_list() (sage.typeset.character_art_factory(CharacterArtFactory method), 394
build_set() (sage.typeset.character_art_factory.CharacterArtFactory method), 395
build_tuple() (sage.typeset.character_art_factory.CharacterArtFactory method), 395
builtin_constant_function() (in module sage.misc.latex), 472
by_opcode (sage.ext.fast_callable.InterpreterMetadata attribute), 347
by_opname (sage.ext.fast_callable.InterpreterMetadata attribute), 347
C
C3_algorithm() (in module sage.misc.c3), 689
C3_merge() (in module sage.misc.c3_controlled), 700
C3_sorted_merge() (in module sage.misc.c3_controlled), 701
c_rand_double() (sage.misc.randstate.randstate method), 73
c_random() (sage.misc.randstate.randstate method), 73
cache (sage.misc.cachefunc.CachedFunction attribute), 276
cache() (sage.misc.sage_input.SageInputBuilder method), 421
cache_key() (in module sage.misc.sagefunc), 297
cached() (sage.misc.cachefunc.CachedFunction method), 276
cached() (sage.misc.cachefunc.CachedMethodCaller method), 285
cached_function() (in module sage.misc.caller method), 285
cached_in_parent_method() (in module sage.misc.cachefunc), 298
cached_method() (in module sage.misc.cachefunc), 300
Cachefunction (class in sage.misc.cachefunc), 274
CacheDict (class in sage.misc.cachefunc), 274
Cachedinparentmethod (class in sage.misc.cachefunc), 280
CachedMethod (class in sage.misc.cachefunc), 281
MethodCaller (class in sage.misc.cachefunc), 284
CachedMethodCallerNoArgs (class in sage.misc.cachefunc), 286
CachedMethodPickle (class in sage.misc.cachefunc), 289
CachedSpecialMethod (class in sage.misc.cachefunc), 290
CachedWeakvalueDictionary (class in sage.misc.weak_dict), 309
delete_rows() (sage.databases.sql_db.SQLDatabase method), 168
denominator() (in module sage.misc.functional), 14
deprecated_function_alias() (in module sage.misc.superseded), 193
DepreciatedFunctionAlias (class in sage.misc.superseded), 193
deprecation() (in module sage.misc.superseded), 194
deprecation_cython() (in module sage.misc.superseded), 195
dest_fd (sage.misc.sage_ostools.redirection attribute), 157
dest_file (sage.misc.sage_ostools.redirection attribute), 157
det() (in module sage.misc.functional), 14
detex() (in module sage.misc.sagedoc), 579
DICT() (sage.misc.explain_pickle.PickleExplaner method), 541
dict() (sage.misc.sage_input.SageInputBuilder method), 423
dict_function() (in module sage.misc.lateX), 474
dict_key() (in module sage.misc.cachefunc), 303
dim() (in module sage.misc.functional), 15
dimension() (in module sage.misc.functional), 15
disc() (in module sage.misc.functional), 16
discriminant() (in module sage.misc.functional), 16
disk_cached_function (class in sage.misc.cachefunc), 304
DiskCachedFunction (class in sage.misc.cachefunc), 291
doc_index() (in module sage.misc.rest_index_of_methods), 684
drop_column() (sage.databases.sql_db.SQLDatabase method), 169
drop_data_from_table() (sage.databases.sql_db.SQLDatabase method), 170
drop_index() (sage.databases.sql_db.SQLDatabase method), 171
drop_primary_key() (sage.databases.sql_db.SQLDatabase method), 171
drop_table() (sage.databases.sql_db.SQLDatabase method), 172
drop_unique() (sage.databases.sql_db.SQLDatabase method), 173
dummy() (sage.misc.nested_class.MainClass.NestedClass.NestedSubClass method), 558
dumps() (in module sage.misc.persit), 523
dumps() (sage.misc.persit.SagePickler class method), 521
DUP() (sage.misc.explain_pickle.PickleExplaner method), 541
dup_source_fd (sage.misc.sage_ostools.redirection attribute), 157
dvi() (sage.misc.latex_standalone.Standalone method), 498
dvi_viewer() (in module sage.misc.viewer), 577
dvi_viewer() (sage.misc.viewer.Viewer method), 575
edit() (in module sage.misc.edit_module), 664
edit_devel() (in module sage.misc.edit_module), 665
EDITOR, 664666
ElementWithLabel (class in sage.misc.element_with_label), 144
ellipsis_iter() (in module sage.arith.range), 138
ellipsis_range() (in module sage.arith.range), 138
empty() (sage.typeset.character_art.CharacterArt class method), 390
EMPTY_DICT() (sage.misc.explain_pickle.PickleExplaner method), 541
EMPTY_LIST() (sage.misc.explain_pickle.PickleExplaner method), 541
empty_subscript() (sage.misc.sage_input.SageInputBuilder method), 423
EMPTY_TUPLE() (sage.misc.explain_pickle.PickleExplaner method), 541
EmptyNewstyleClass (in sage.misc.explain_pickle), 541
EmptyOldstyleClass (in sage.misc.explain_pickle), 541
engine() (sage.misc.latex.Latex method), 462
ensure_startup_finished() (in module sage.misc.lazy_import), 245
environment variable
EDITOR, 664666
PATH, 154
eps() (sage.misc.latex_standalone.Standalone method), 499
eva() (in module sage.misc.functional), 16
eval() (sage.misc.html.HTMLFragmentFactory method), 451
eval() (sage.misc.html.MathJax method), 454
eval() (sage.misc:inline_fortran.InlineFortran method), 360
eval() (sage.misc.latex.Latex method), 462
eval() (sage.misc.python.Python method), 378
eval() (sage.misc.sage_timeit_class.SageTimeit method), 631
eval() (sage.misc.sh.Sh method), 384
exactly_one_is_true() (in module sage.misc), 199
exists() (in module sage.misc), 200
experimental (class in sage.misc.superseded), 195
experimental_warning() (in module sage.misc.superseded), 196
explain_pickle() (in module sage.misc.explain_pickle), 550
explain_pickle_string() (in module sage.misc.explain_pickle), 552
exponent() (sage.ext.fast_callable.ExpressionIPow method), 335
expovariate() (in module sage.misc.prandom), 79
Expression (class in sage.ext.fast_callable), 330
ExpressionCall (class in sage.ext.fast_callable), 331
ExpressionChoice (class in sage.ext.fast_callable), 332
ExpressionConstant (class in sage.ext.fast_callable), 334
ExpressionIPow (class in sage.ext.fast_callable), 335
ExpressionTreeBuilder (class in sage.ext.fast_callable), 336
ExpressionVariable (class in sage.ext.fast_callable), 339
EXT1() (sage.misc.explain_pickle.PickleExplainer method), 541
EXT2() (sage.misc.explain_pickle.PickleExplainer method), 542
EXT4() (sage.misc.explain_pickle.PickleExplainer method), 542
extend() (sage.misc.explain_pickle.TestAppendList method), 548
extra_macros() (sage.misc.latex.Latex method), 463
extra_preamble() (sage.misc.latex.Latex method), 463

F
f (sage.misc.cachefunc.CachedFunction attribute), 277
f (sage.misc.function_mangling.ArgumentFixer attribute), 682
fast_callable() (in module sage.ext.fast_callable), 348
fast_float() (in module sage.ext.fast_eval), 355
FastCallableFloatWrapper (class in sage.ext.fast_callable), 339
FastHashable_class (class in sage.misc.fast_methods), 116
fcp() (in module sage.misc.functional), 17
file_and_line() (in module sage.misc.edit_module), 666
file_list() (sage.misc.cachefunc.FileCache method), 293
FileCache (class in sage.misc.cachefunc), 292
filename() (sage.misc.gperftools.Profiler method), 635
find_object_modules() (in module sage.misc.dev_tools), 669
find_objects_from_name() (in module sage.misc.dev_tools), 670
finish_startup() (in module sage.misc.lazy_import), 245
fix_to_named() (sage.misc.function_mangling.ArgumentFixer method), 682
fix_to_pos() (sage.misc.function_mangling.ArgumentFixer method), 683
flatten() (in module sage.misc.flatten), 125
FLOAT() (sage.misc.explain_pickle.PickleExplainer method), 542
float_function() (in module sage.misc.latex), 474
float_str() (sage.misc.sage_input.SageInputBuilder method), 424
foo() (in module sage.misc.parser), 377
forall() (in module sage.misc.sageinspect), 201
format() (in module sage.misc.sagedoc), 580
format() (sage.misc.sage_input.SageInputFormatter method), 435
format_search_as_html() (in module sage.misc.sagedoc), 581
format_src() (in module sage.misc.sagedoc), 582
formatannotation() (in module sage.misc.sageinspect), 651
from_dot_string() (sage.misc.latex_standalone.TikzPicture class method), 508
from_graph() (sage.misc.latex_standalone.TikzPicture class method), 510
from_graph_with_pos() (sage.misc.latex_standalone.TikzPicture class method), 514
from_poset() (sage.misc.latex_standalone.TikzPicture class method), 515
func_persist (class in sage.misc.func_persist), 519
function() (sage.ext.fast_callable.ExpressionCall method), 332
function_name() (in module sage.ext.fast_callable), 352

G
gammavariate() (in module sage.misc.prandom), 80
gauss() (in module sage.misc.prandom), 80
gen() (in module sage.misc.functional), 17
gen() (sage.misc.sage_input.SageInputBuilder method), 424
gen_rest_table_index() (in module sage.misc.rest_index_of_methods), 684
gen_thematic_rest_table_index() (in module sage.misc.rest_index_of_methods), 686
generate_code() (in module sage.ext.fast_callable), 352
gens() (in module sage.misc.functional), 18
GET() (sage.misc.explain_pickle.PickleExplainer method), 542
get() (sage.misc.lazy_list.lazy_list_generic method), 258
get() (sage.misc.sagein_psect), 312
get_baseline() (sage.typeset.character_art.CharacterArt method), 390
get_breakpoints() (sage.typeset.character_art.CharacterArt method), 390
get_builtn functions() (in module sage.ext.fast_callable), 353
get_cache_file() (in module sage.misc.import_cache), 249
get_connection() (sage.databases.sql_db.SQLDatabase method), 173
get_current() (sage.ext.fast_callable.InstructionStream method), 341
get_cursor() (sage.databases.sql_db.SQLDatabase method), 174
get_key() (sage.misc.cachefunc.CachedFunction method), 277
get_main_globals() (in module sage.misc.misc), 202
get_metadata() (sage.ext.fast_callable.InstructionStream method), 342
get_name() (sage.misc.sage_input.SageInputFormatter method), 436
get_orig_args() (sage.ext.fast_callable.Wrapper method), 347
get_query_string() (sage.databases.sql_db.SQLQuery method), 183
get_remote_file() (in module sage.misc.remote_file), 152
get_skeleton() (sage.databases.sql_db.SQLDatabase method), 175
get_star_imports() (in module sage.misc.lazy_import), 245
get_syst ems() (in module sage.misc.citation), 604
get_verbose() (in module sage.misc.verbose), 226
get_verbose_files() (in module sage.misc.verbose), 226
getattr() (sage.misc.sage_input.SageInputBuilderFactory method), 425
getrandbits() (in module sage.misc.prandom), 81
GLOBAL() (sage.misc.explain_pickle.PickleExplainer method), 542
GloballyCachedMethodCaller (class in sage.misc.cachefunc), 295
html() (in module sage.misc.html), 455
HtmlFragment (class in sage.misc.html), 453
HTMLFragmentFactory (class in sage.misc.html), 451
id_cache() (sage.misc.sage_input.SageInputBuilderFactory method), 425
identity() (in module sage.misc.c3_controlled), 717
if_false() (sage.ext.fast_callable.ExpressionChoice method), 333
if_true() (sage.ext.fast_callable.ExpressionChoice method), 334
iframe() (sage.misc.html.HTMLFragmentFactory method), 452
image() (in module sage.misc.functional), 19
import_name() (sage.misc.sage_input.SageInputBuilderFactory method), 426
import_statement_string() (in module sage.misc.dev_tools), 671
import_statements() (in module sage.misc.dev_tools), 671
increase_recursion_limit() (in module sage.misc.misc), 203
infix_operator (class in sage.misc.decorators), 101
info() (sage.misc.sage_unittest.InstanceTester method), 607
InheritComparisonClasscallMeta class (class in sage.misc.inherit_comparison), 114
InheritComparisonMeta class (class in sage.misc.inherit_comparison), 114
init() (in module sage.misc.session), 563
init_memory_functions() (in module sage.ext.memory), 689
initial_seed() (in module sage.misc.randstate), 72
inject_variable() (in module sage.misc.misc), 203
inject_variable_test() (in module sage.misc.misc), 205
InlineFortran (class in sage.misc.inline_fortran), 360
Inner2 (class in sage.misc.bindable_class), 99
Inner2 (sage.misc.bindable_class,Outer attribute), 99
INST() (sage.misc.explain_pickle.PickleExplainer method), 542
installed_packages() (in module sage.misc.package), 596
installed_version (sage.misc.package.PackageInfo attribute), 596
instance_tester() (in module sage.misc.sage_unittest), 618
instancetoc() (in module sage.misc.instancetoc), 641
InstanceDocDescriptor (class in sage.misc.instancetoc), 640
InstanceTester (class in sage.misc.sage_unittest), 607
inst() (sage.ext.fast_callable.InstructionStream method), 343

H
has_file() (sage.misc.latex.Latex method), 464
has_instr() (sage.ext.fast_callable.InstructionStream method), 343
has_latex attr() (in module sage.misc.latex), 475
have_program() (in module sage.misc.sage_ostools), 154
hecke_operator() (in module sage.misc.functional), 18
height() (sage.typeset.character_art.CharacterArt method), 391
help() (in module sage.misc.sagedoc), 583
HierarchyElement (class in sage.misc.c3_controlled), 707

Index
<table>
<thead>
<tr>
<th>Function/Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstructionStream (class in sage.ext.fast_callable), 340</td>
</tr>
<tr>
<td>INT () (sage.misc.explain_pickle.PickleExplainer method), 542</td>
</tr>
<tr>
<td>int () (sage.misc.sage_input.SageInputBuilder method), 427</td>
</tr>
<tr>
<td>IntegerPowerFunction (class in sage.ext.fast_callable), 345</td>
</tr>
<tr>
<td>integral () (module sage.misc.functional), 19</td>
</tr>
<tr>
<td>integral_closure() (module sage.misc.functional), 21</td>
</tr>
<tr>
<td>integrate() (module sage.misc.functional), 22</td>
</tr>
<tr>
<td>InterpreterMetadata (class in sage.ext.fast_callable), 346</td>
</tr>
<tr>
<td>intersect() (sage.databases.sql_db.SQLQuery method), 183</td>
</tr>
<tr>
<td>interval() (module sage.misc.functional), 24</td>
</tr>
<tr>
<td>ipow_range (sage.ext.fast_callable.InterpreterMetadata attribute), 347</td>
</tr>
<tr>
<td>is_duration_startup() (in module sage.misc.lazy_import), 245</td>
</tr>
<tr>
<td>is_even() (module sage.misc.functional), 24</td>
</tr>
<tr>
<td>is_fast_float() (module sage.ext.fast_eval), 356</td>
</tr>
<tr>
<td>is_function_or_cython_function() (module sage.misc.sageinspect), 652</td>
</tr>
<tr>
<td>is_in_cache() (sage.misc.cachefunc.CachedFunction method), 279</td>
</tr>
<tr>
<td>is_in_cache() (sage.misc.cachefunc.CachedMethod-CallerNoArgs method), 287</td>
</tr>
<tr>
<td>is_in_string() (module sage.misc.misc), 205</td>
</tr>
<tr>
<td>is_installed() (sage.misc.package.PackageInfo method), 596</td>
</tr>
<tr>
<td>is_iterator() (module sage.misc.misc), 205</td>
</tr>
<tr>
<td>is_lazy_string() (module sage.misc.lazy_string), 260</td>
</tr>
<tr>
<td>is_mutable_pickle_object() (sage.misc.explain_pickle.PickleExplainer method), 543</td>
</tr>
<tr>
<td>is_odd() (module sage.misc.functional), 24</td>
</tr>
<tr>
<td>is_optional() (sage.misc.abstract_method.AbstractMethod method), 91</td>
</tr>
<tr>
<td>is_package_installed() (module sage.misc.package), 597</td>
</tr>
<tr>
<td>is_package_installed_and_updated() (module sage.misc.package), 597</td>
</tr>
<tr>
<td>isclassinstance() (module sage.misc.misc), 207</td>
</tr>
<tr>
<td>isqrt() (module sage.misc.functional), 25</td>
</tr>
<tr>
<td>items() (sage.misc.cachefunc.FileCache method), 293</td>
</tr>
<tr>
<td>items() (sage.misc.weak_dict.WeakValueDictionary method), 313</td>
</tr>
<tr>
<td>items_list() (sage.misc.weak_dict.WeakValueDictionary method), 314</td>
</tr>
<tr>
<td>iterator_prod() (module sage.misc.misc.c), 215</td>
</tr>
<tr>
<td>iterable() (sage.misc.weak_dict.WeakValueDictionary method), 316</td>
</tr>
<tr>
<td>iteritems() (sage.misc.weak_dict.WeakValueDictionary method), 316</td>
</tr>
<tr>
<td>itervalues() (sage.misc.weak_dict.WeakValueDictionary method), 316</td>
</tr>
<tr>
<td>kernel() (module sage.misc.functional), 25</td>
</tr>
<tr>
<td>KeyConvertingDict (class in sage.misc.converting_dict), 122</td>
</tr>
<tr>
<td>keys() (sage.misc.cachefunc.FileCache method), 294</td>
</tr>
<tr>
<td>keys() (sage.misc.weak_dict.WeakValueDictionary method), 317</td>
</tr>
<tr>
<td>krull_dimension() (module sage.misc.functional), 26</td>
</tr>
<tr>
<td>last() (sage.misc.parser.Tokenizer method), 374</td>
</tr>
<tr>
<td>last_token_string() (sage.misc.parser.Tokenizer method), 375</td>
</tr>
<tr>
<td>Latex (class in sage.misc.latex), 458</td>
</tr>
<tr>
<td>latex() (module sage.misc.latex), 475</td>
</tr>
<tr>
<td>latex_extra_preamble() (module sage.misc.latex), 476</td>
</tr>
<tr>
<td>latex_variable_name() (module sage.misc.latex), 477</td>
</tr>
<tr>
<td>latex_variable_names() (module sage.misc.defaults), 1</td>
</tr>
<tr>
<td>latex_varify() (module sage.misc.latex), 479</td>
</tr>
<tr>
<td>LatexCall (class in sage.misc.latex), 468</td>
</tr>
<tr>
<td>LatexExamples (class in sage.misc.latex), 468</td>
</tr>
<tr>
<td>LatexExamples.png (class in sage.misc.latex), 469</td>
</tr>
<tr>
<td>LatexExamples.png (class in sage.misc.latex), 469</td>
</tr>
<tr>
<td>LatexExamples.pstricks (class in sage.misc.latex), 470</td>
</tr>
<tr>
<td>LatexExpr (class in sage.misc.latex), 470</td>
</tr>
<tr>
<td>lazy_attribute (class in sage.misc.lazy_attribute), 227</td>
</tr>
<tr>
<td>lazy_class_attribute (class in sage.misc.lazy_attribute), 237</td>
</tr>
<tr>
<td>lazy_import() (module sage.misc.lazy_attribute), 245</td>
</tr>
<tr>
<td>lazy_list() (module sage.misc.lazy_list), 253</td>
</tr>
<tr>
<td>lazy_list_formatter() (module sage.misc.lazy_list), 254</td>
</tr>
<tr>
<td>lazy_list_from_function (class in sage.misc.lazy_list), 256</td>
</tr>
<tr>
<td>lazy_list_from_iterator (class in sage.misc.lazy_list), 256</td>
</tr>
<tr>
<td>lazy_list_from_update_function (class in sage.misc.lazy_list), 257</td>
</tr>
<tr>
<td>lazy_list_generic (class in sage.misc.lazy_list), 257</td>
</tr>
<tr>
<td>lazy_string() (module sage.misc.lazy_string), 261</td>
</tr>
</tbody>
</table>
LazyFormat (class in sage.misc.lazy_format), 240
LazyImport (class in sage.misc.lazy_import), 243
License (class in sage.misc.copying), 605
lift() (in module sage.misc.functional), 27
LIST() (sage.misc.explain_pickle.PickleExplainer method), 542
list() (sage.misc.lazy_list.lazy_list_generic method), 259
list_function() (in module sage.misc.functional), 27
list() (sage.misc.lazy_list.lazy_list_generic method), 259
list_of_subfunctions() (in module sage.misc.rest_index_of_methods), 687
list_packages() (in module sage.misc.package), 598
load() (in module sage.misc.persist), 523
load_arg() (sage.ext.fast_callable.InstructionStream method), 344
load_const() (sage.ext.fast_callable.InstructionStream method), 345
load_sage_element() (in module sage.misc.persist), 525
load_sage_object() (in module sage.misc.persist), 526
load_session() (in module sage.misc.session), 563
load_submodules() (in module sage.misc.dev_tools), 678
loads() (in module sage.misc.persist), 526
loads() (sage.misc.persist.SageUnpicker class method), 522
log() (in module sage.misc.functional), 27
lognormvariate() (in module sage.misc.prandom), 81
LONG() (sage.misc.explain_pickle.PickleExplainer method), 542
LONG1() (sage.misc.explain_pickle.PickleExplainer method), 542
LONG4() (sage.misc.explain_pickle.PickleExplainer method), 542
LONG_BINGET() (sage.misc.explain_pickle.PickleExplainer method), 542
LONG_BINPUT() (sage.misc.explain_pickle.PickleExplainer method), 542
long_seed() (sage.misc.randstate.randstate method), 74
longMessage (sage.misc.sage_unittest.InstanceTester attribute), 608
LookupNameMaker (class in sage.misc.parser), 362

M

MainClass (class in sage.misc.nested_class), 557
MainClass.NestedClass (class in sage.misc.nested_class), 557
MainClass.NestedClass.NestedSubClass (class in sage.misc.nested_class), 558
make_index() (sage.databases.sql_db.SQLDatabase method), 176
make_None() (in module sage.misc.persist), 527
make_primary_key() (sage.databases.sql_db.SQLDatabase method), 176
make_unique() (sage.databases.sql_db.SQLDatabase method), 177
map_threaded() (in module sage.misc.map_threaded), 137
MARK() (sage.misc.explain_pickle.PickleExplainer method), 542
math_parse() (in module sage.misc.html), 456
MathJax (class in sage.misc.html), 453
MathJaxExpr (class in sage.misc.html), 454
MathML (class in sage.misc.html), 457
mathml() (in module sage.misc.mathml), 457
matrix_column_alignment() (sage.misc.latex.LaTeX method), 465
matrix_delimiters() (sage.misc.latex.LaTeX method), 466
MethodDecorator (class in sage.misc.method_decorator), 115
minimal_polynomial() (in module sage.misc.functional), 32
minpoly() (in module sage.misc.functional), 32
modify_for_nested_pickle() (in module sage.misc.nested_class), 559
module
sage.arith.srange, 138
sage.databases.sql_db, 158
sage.ext.fast_callable, 322
sage.ext.fast_eval, 355
sage.ext.memory, 689
sage.misc.abstract_method, 91
sage.misc.banner, 569
sage.misc.benchmark, 623
sage.misc.bindable_class, 96
sage.misc.c3, 689
sage.misc.c3_controlled, 691
sage.misc.cachefunc, 262
sage.misc.call, 119
sage.misc.callable_dict, 120
sage.misc.citation, 603
sage.misc.classcall_metaclass, 109
sage.misc.classgraph, 668
sage.misc.constant_function, 108
sage.misc.converting_dict, 121
sage.misc.copying, 605
sage.misc.cython, 357
sage.misc.decorators, 99
sage.misc.defaults, 1
sage.misc.dev_tools, 669
sage.misc.edit_module, 664
sage.misc.element_with_label, 144
sage.misc.explain_pickle, 538
sage.misc.fast_methods, 116
sage.misc.fpickle, 536
sage.misc.func_persist, 519
sage.misc.function_mangling, 680
sage.misc.functional, 3
sage.misc.gperftools, 634
sage.misc.html, 451
sage.misc.inherit_comparison, 113
sage.misc.inline_fortran, 360
sage.misc.instancedoc, 638
sage.misc.latex, 458
sage.misc.latex_macros, 485
sage.misc.latex_standalone, 487
sage.misc.lazy_attribute, 227
sage.misc.lazy_format, 240
sage.misc.lazy_import, 242
sage.misc.lazy_import_cache, 249
sage.misc.lazy_list, 250
sage.misc.lazy_string, 260
sage.misc.map_threaded, 137
sage.misc.mathml, 457
sage.misc.messaging, 153
sage.misc.method_decorator, 115
sage.misc.misc, 197
sage.misc.misc_c, 213
sage.misc.mrange, 127
sage.misc.multireplace, 136
sage.misc.nested_class, 555
sage.misc.object_multiplexer, 116
sage.misc.package, 595
sage.misc.parser, 362
sage.misc.persist, 519
sage.misc.prandom, 77
sage.misc.profiler, 632
sage.misc.python, 378
sage.misc.random_testing, 619
sage.misc.randstate, 63
sage.misc.remote_file, 152
sage.misc.repr, 407
sage.misc.reset, 572
sage.misc.rest_index_of_methods, 684
sage.misc.sage_eval, 380
sage.misc.sage_input, 410
sage.misc.sage_ostools, 154
sage.misc.sage_timeit, 627
sage.misc.sage_timeit_class, 631
sage.misc.sage_unittest, 607
sage.misc.sagedoc, 579
sage.misc.sagedoc_conf, 594
sage.misc.sageinspect, 641
sage.misc.search, 127
sage.misc.session, 561
sage.misc.sh, 384
sage.misc.sphinxify, 594
sage.misc.stopgap, 191
sage.misc.superseded, 193
sage.misc.table, 441
sage.misctemporary_file, 145
sage.misc.test_nested_class, 622
sage.misc.unknown, 86
sage.misc.verbose, 224
sage.miscviewer, 574
sage.misc.weak_dict, 306
sage.typeset.ascii_art, 396
sage.typeset.character_art, 389
sage.typeset.character_art_factory, 392
sage.typeset.symbols, 385
sage.typeset.unicode_art, 402
sage.typeset.unicode_characters, 405
mrange() (in module sage.misc.mrange), 129
mrange_iter() (in module sage.misc.mrange), 130
mro() (sage.misc.c3_controlled.HierarchyElement method), 715
mro_controlled() (sage.misc.c3_controlled.HierarchyElement method), 716
mro_standard() (sage.misc.c3_controlled.HierarchyElement method), 716
multiple_replace() (in module sage.misc.multireplace), 136
Multiplex (class in sage.misc.object_multiplexer), 116
MultiplexFunction (class in sage.misc.object_multiplexer), 116
multiplicative_order() (in module sage.misc.functional), 33
my_getsource() (in module sage.misc.sagedoc), 583

N
N() (in module sage.misc.functional), 3
n() (in module sage.misc.functional), 33
name (sage.misc.package.PackageInfo attribute), 596
name() (sage.misc.sage_input.SageInputBuilder method), 428
name_is_valid() (in module sage.misc.explain_pickle), 552
nest() (in module sage.misc.misc), 207
nested_pickle() (in module sage.misc.nested_class), 560
NestedClassMetaclass (class in sage.misc.nested_class), 560
NEWFALSE() (sage.misc.explain_pickle.PickleExplainer method), 558
NEWOBJ() (sage.misc.explain_pickle.PickleExplainer method), 542
Utilities, Release 10.4

newton_method_sizes()  (in module sage.misc.misc), 208
NEWTRUE()  (sage.misc.explain_pickle.PickleExplainer method), 542
next()  (sage.misc.parser.Tokenizer method), 375
ngens()  (in module sage.misc.functional), 38
NonAssociative  (class in sage.misc.misc_c), 213
NONE()  (sage.misc.explain_pickle.PickleExplainer method), 542
None_function()  (in module sage.misc.latex), 472
NonpicklingDict  (class in sage.misc.cachefunc), 295
norm()  (in module sage.misc.functional), 39
normalize_index()  (in module sage.misc.misc_c), 216
normalvariate()  (in module sage.misc.prandom), 81
numerator()  (in module sage.misc.functional), 42
numerical_approx()  (in module sage.misc.functional), 42

O
OBJ()  (sage.misc.explain_pickle.PickleExplainer method), 542
objgen()  (in module sage.misc.functional), 47
objgens()  (in module sage.misc.functional), 48
op_list()  (in module sage.ext.fast_callable), 354
op_list()  (sage.ext.fast_callable.Wrapper method), 347
operators  (sage.misc.decorators.infix_operator_attribute), 103
options()  (class in sage.misc.decorators), 103
order()  (in module sage.misc.functional), 48
Outer  (class in sage.misc.bindable_class), 99
Outer.Inner  (class in sage.misc.bindable_class), 99

P
p_arg()  (sage.misc.parser.Parser method), 364
p_args()  (sage.misc.parser.Parser method), 364
p_atom()  (sage.misc.parser.Parser method), 365
p_eqn()  (sage.misc.parser.Parser method), 365
p_expr()  (sage.misc.parser.Parser method), 366
p_factor()  (sage.misc.parser.Parser method), 367
p_list()  (sage.misc.parser.Parser method), 367
p_matrix()  (sage.misc.parser.Parser method), 368
p_power()  (sage.misc.parser.Parser method), 368
p_sequence()  (sage.misc.parser.Parser method), 369
p_term()  (sage.misc.parser.Parser method), 370
p_tuple()  (sage.misc.parser.Parser method), 370
package_manifest()  (in module sage.misc.package), 599
package_versions()  (in module sage.misc.package), 600
PackageInfo  (class in sage.misc.package), 596
pad_zeros()  (in module sage.misc.misc), 209
pager()  (in module sage.misc.pager), 579
parent_with_gens()  (sage.misc.sage_input.SageInputBuilder method), 428
paretovariate()  (in module sage.misc.prandom), 82
parse()  (sage.misc.parser.Parser method), 370
parse_expression()  (sage.misc.parser.Parser method), 371
parse_keywords()  (sage.typeset.character_art_factory.CharacterArtFactory method), 395
parse_sequence()  (sage.misc.parser.Parser method), 371
Parser  (class in sage.misc.parser), 362
PATH, 154
df()  (sage.misc.latex_standalone.Standalone method), 500
df_viewer()  (in module sage.misc.viewer), 578
df_viewer()  (sage.misc.viewer.Viewer method), 576
peek()  (sage.misc.parser.Tokenizer method), 376
PERSID()  (sage.misc.explain_pickle.PickleExplainer method), 542
pickle_function()  (in module sage.misc.fpickle), 537
PickleDict  (class in sage.misc.explain_pickle), 541
PickleExplainer  (class in sage.misc.explain_pickle), 541
PickleInstance  (class in sage.misc.explain_pickle), 547
picklejar()  (in module sage.misc.persist), 528
pickleMethod()  (in module sage.misc.fpickle), 536
pickleModule()  (in module sage.misc.fpickle), 537
PickleObject  (class in sage.misc.explain_pickle), 547
pip_installed_packages()  (in module sage.misc.package), 601
pip_remote_version()  (in module sage.misc.package), 602
pkgname_split()  (in module sage.misc.package), 603
png()  (in module sage.misc.latex), 480
png()  (sage.misc.latex_standalone.Standalone method), 501
png_viewer()  (in module sage.misc.viewer), 578
png_viewer()  (sage.misc.viewer.Viewer method), 576
pop()  (sage.misc.converting_dict.KeyConvertingDict method), 123
POP()  (sage.misc.explain_pickle.PickleExplainer method), 542
pop()  (sage.misc.explain_pickle.PickleExplainer method), 544
pop()  (sage.misc.weak_dict.WeakValueDictionary method), 317
POP_MARK()  (sage.misc.explain_pickle.PickleExplainer method), 542
pop_to_mark()  (sage.misc.explain_pickle.PickleExplainer method), 544
popitem()  (sage.misc.weak_dict.WeakValueDictionary
Utilities, Release 10.4

run_pickle() (sage.misc.explain_pickle.PickleEx-
plainer method), 546
running_total() (in module sage.misc.misc_c), 221
runsnake() (in module sage.misc.dev_tools), 679
runTest() (sage.misc.sageunittest.InstanceTester
method), 608

S
sage_eval() (in module sage.misc.sage_eval), 380
sage_formatargspec() (in module sage.misc.sagein-
spect), 653
sage_getargspec() (in module sage.misc.sagein-
spect), 654
sage_getdef() (in module sage.misc.sageinspect), 656
sage_getdoc() (in module sage.misc.sageinspect), 657
sage_getdoc_original() (in module sage.misc.sageinspect), 658
sage_getfile() (in module sage.misc.sageinspect), 660
sage_getfile_relative() (in module sage.misc.sageinspect), 661
sage_getsource() (in module sage.misc.sageinspect), 662
sage_getsourceclines() (in module sage.misc.sageinspect), 663
sage_getvariablename() (in module sage.misc.sageinspect), 663
sage_input() (in module sage.misc.sage_input), 437
sage_latex_macros() (in module sage.misc.lat-

tex_macros), 487
sage_mathjax_macros() (in module sage.misc.lat-

tex_macros), 487
sage_timeit() (in module sage.misc.sage_timeit), 628
sage_wraps() (in module sage.misc.decorators), 103
SageArgSpecVisitor (class in sage.misc.sagein-
spect), 645
sage.arith.srange
module, 138
sage.databases.sql_db
module, 158
sage.ext.fast_callable
module, 322
sage.ext.fast_eval
module, 355
sage.ext.memory
module, 689
SageInputAnswer (class in sage.misc.sage_input), 419
SageInputBuilder (class in sage.misc.sage_input), 420
SageInputExpression (class in sage.misc.sage_input), 434
SageInputFormatter (class in sage.misc.sage_input), 435
SagemathTransform (class in sage.misc.sage-
doc_conf), 594
sage.misc.abstract_method
module, 91
sage.misc.banner
module, 569
sage.misc.benchmark
module, 623
sage.misc.bindable_class
module, 96
sage.misc.c3
module, 689
sage.misc.c3_controlled
module, 691
sage.misc.cachefunc
module, 262
sage.misc.call
module, 119
sage.misc.callable_dict
module, 120
sage.misc.citation
module, 603
sage.misc.classcall_metaclass
module, 109
sage.misc.classgraph
module, 668
sage.misc.constant_function
module, 108
sage.misc.conveting_dict
module, 121
sage.misc.copying
module, 605
sage.misc.cython
module, 357
sage.misc.decorators
module, 99
sage.misc.defaults
module, 1
sage.misc.dev_tools
module, 669
sage.misc.edit_module
module, 664
sage.misc.element_with_label
module, 144
sage.misc.explain_pickle
module, 538
sage.misc.fast_methods
module, 116
sage.misc.flatten
module, 125
sage.misc.fpickle
module, 536
sage.misc.func_persist
module, 519
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage.misc.function_mangling</td>
<td>680</td>
</tr>
<tr>
<td>sage.misc.functional</td>
<td>3</td>
</tr>
<tr>
<td>sage.misc.gperftools</td>
<td>634</td>
</tr>
<tr>
<td>sage.misc.html</td>
<td>451</td>
</tr>
<tr>
<td>sage.misc.inherit_comparison</td>
<td>113</td>
</tr>
<tr>
<td>sage.misc.inline_fortran</td>
<td>360</td>
</tr>
<tr>
<td>sage.misc.instancedoc</td>
<td>638</td>
</tr>
<tr>
<td>sage.misc.latex</td>
<td>458</td>
</tr>
<tr>
<td>sage.misc.latex_macros</td>
<td>485</td>
</tr>
<tr>
<td>sage.misc.latex_standalone</td>
<td>487</td>
</tr>
<tr>
<td>sage.misc.lazy_attribute</td>
<td>227</td>
</tr>
<tr>
<td>sage.misc.lazy_format</td>
<td>240</td>
</tr>
<tr>
<td>sage.misc.lazy_import</td>
<td>242</td>
</tr>
<tr>
<td>sage.misc.lazy_import_cache</td>
<td>249</td>
</tr>
<tr>
<td>sage.misc.lazy_list</td>
<td>250</td>
</tr>
<tr>
<td>sage.misc.lazy_string</td>
<td>260</td>
</tr>
<tr>
<td>sage.misc.map_threaded</td>
<td>137</td>
</tr>
<tr>
<td>sage.misc.mathml</td>
<td>457</td>
</tr>
<tr>
<td>sage.misc.messaging</td>
<td>153</td>
</tr>
<tr>
<td>sage.misc.method_decorator</td>
<td>115</td>
</tr>
<tr>
<td>sage.misc.misc</td>
<td>197</td>
</tr>
<tr>
<td>sage.misc.misc_c</td>
<td>213</td>
</tr>
<tr>
<td>sage.misc.mrange</td>
<td>127</td>
</tr>
<tr>
<td>sage.misc.multireplace</td>
<td>136</td>
</tr>
<tr>
<td>sage.misc.nested_class</td>
<td>555</td>
</tr>
<tr>
<td>sage.misc.object_multiplexer</td>
<td>116</td>
</tr>
<tr>
<td>sage.misc.package</td>
<td>595</td>
</tr>
<tr>
<td>sage.misc.pager</td>
<td>579</td>
</tr>
<tr>
<td>sage.misc.parser</td>
<td>362</td>
</tr>
<tr>
<td>sage.misc.persist</td>
<td>519</td>
</tr>
<tr>
<td>sage.misc.prandom</td>
<td>77</td>
</tr>
<tr>
<td>sage.misc.profiler</td>
<td>632</td>
</tr>
<tr>
<td>sage.misc.python</td>
<td>378</td>
</tr>
<tr>
<td>sage.misc.random_testing</td>
<td>619</td>
</tr>
<tr>
<td>sage.misc.randstate</td>
<td>63</td>
</tr>
<tr>
<td>sage.misc.remote_file</td>
<td>152</td>
</tr>
<tr>
<td>sage.misc.repr</td>
<td>407</td>
</tr>
<tr>
<td>sage.misc.reset</td>
<td>572</td>
</tr>
<tr>
<td>sage.misc.rest_index_of_methods</td>
<td>684</td>
</tr>
<tr>
<td>sage.misc.sage_eval</td>
<td>380</td>
</tr>
<tr>
<td>sage.misc.sage_input</td>
<td>410</td>
</tr>
<tr>
<td>sage.misc.sage_ostools</td>
<td>154</td>
</tr>
<tr>
<td>sage.misc.sage_timeit</td>
<td>627</td>
</tr>
<tr>
<td>sage.misc.sage_timeit_class</td>
<td>631</td>
</tr>
<tr>
<td>sage.misc.sage_unittest</td>
<td>607</td>
</tr>
<tr>
<td>sage.misc.sagedoc</td>
<td>579</td>
</tr>
<tr>
<td>sage.misc.sagedoc_conf</td>
<td>594</td>
</tr>
<tr>
<td>sage.misc.sageinspect</td>
<td>641</td>
</tr>
<tr>
<td>sage.misc.search</td>
<td>127</td>
</tr>
<tr>
<td>sage.misc.session</td>
<td>561</td>
</tr>
<tr>
<td>sage.misc.sh</td>
<td>384</td>
</tr>
<tr>
<td>sage.misc.sphinxify</td>
<td>594</td>
</tr>
<tr>
<td>sage.misc.stopgap</td>
<td>191</td>
</tr>
<tr>
<td>sage.misc.superseded</td>
<td>193</td>
</tr>
</tbody>
</table>
Utilities, Release 10.4

sage.misc.table
module, 441
sage.misc.temporary_file
module, 145
sage.misc.test_nested_class
module, 622
sage.misc.unknown
module, 86
sage.misc.verbose
module, 224
sage.misc.viewer
module, 574
sage.misc.weak_dict
module, 306
sageobj() (in module sage.misc.sage_eval), 383
SagePickle (in module sage.misc.persister), 519
SageTimeit (in module sage.misc.sage_timeit_class), 631
SageTimeitResult (in module sage.misc.sage_timeit),
627
sage.typeset.ascii_art
module, 396
sage.typeset.character_art
module, 389
sage.typeset.character_art_factory
module, 392
sage.typeset.symbols
module, 385
sage.typeset.unicode_art
module, 402
sage.typeset.unicode_characters
module, 405
SageUnpickler (in module sage.misc.persister), 521
sample() (in module sage.misc.prandom), 84
sanitize() (in module sage.misc.cython), 360
save() (in module sage.misc.persister), 533
save() (sage.databases.sql_db.SQLDatabase method),
179
save() (sage.misc.sage_input.Profiler method), 635
save() (sage.misc.sage_input.Profiler method), 635
save_cache_file() (in module sage.misc.lazy_import), 249
save_session() (in module sage.misc.sagedoc_session),
564
search() (in module sage.misc.sage_search), 127
search_def() (in module sage.misc.sagedoc), 586
search_doc() (in module sage.misc.sagedoc), 586
search_src() (in module sage.misc.sagedoc), 587
seed (in module sage.misc.randstate), 77
seed() (sage.misc.randstate.randstate method), 74
series_precision() (in module sage.misc.defaults),
1
set_cache() (sage.misc.cachefunc.CachedMethod-
CallerNoArgs method), 288
set_default_variable_name() (in module sage.misc.defaults), 2
set_edit_template() (in module sage.misc.edit_module), 666
set_editor() (in module sage.misc.edit_module), 667
set_names() (sage.misc.parser.LookupNameMaker
method), 362
set_random_seed() (in module sage.misc.randstate),
77
set_seed_gap() (sage.misc.randstate.randstate
method), 75
set_seed_gp() (sage.misc.randstate.randstate
method), 75
set_seed_libc() (sage.misc.randstate.randstate
method), 76
set_seed_ntl() (sage.misc.randstate.randstate
method), 76
set_seed_pari() (sage.misc.randstate.randstate
method), 76
set_series_precision() (in module sage.misc.def-
defaults), 2
set_state() (in module sage.misc.stopgap), 191
set_verbose() (in module sage.misc.verbose), 226
set_verbose_files() (in module sage.misc.verbose), 227
setdefault() (sage.misc.converting_dict.KeyConver-
ingDict method), 123
setdefault() (sage.misc.collections.Dict
method), 319
SETITEM() (sage.misc.explain_pickle.PickleExplainer
method), 542
SETITEMS() (sage.misc.explain_pickle.PickleExplainer
method), 542
setup() (in module sage.misc.sagedoc_conf), 594
Sh (in class sage.misc.sh), 384
share() (sage.misc.sage_input.Profiler method), 547
share() (sage.misc.sage_input.Profiler method), 547
share() (sage.misc.sage_input.Profiler method), 547
SHORT_BINSTRING() (sage.misc.explain_pickle.Pro-
fileExplainer method), 432
show() (sage.databases.sql_db.SQLDatabase method),
180
show() (sage.databases.sql_db.SQLQuery method), 185
show_identifiers() (in module sage.misc.sagedoc_session),
566
shuffle() (in module sage.misc.prandom), 85
SIE_assign (in module sage.misc.sage_input), 414
SIE_binary (in module sage.misc.sage_input), 414
SIE_call (in module sage.misc.sage_input), 415
SIE_dict (in module sage.misc.sage_input), 415
SIE_gen (in module sage.misc.sage_input), 416
transpose() (sage.misc.table.table method), 450
try_read() (in module sage.misc.misc), 210
TUPLE() (sage.misc.explain_pickle.PickleExplainer method), 542
TUPLE1() (sage.misc.explain_pickle.PickleExplainer method), 543
TUPLE2() (sage.misc.explain_pickle.PickleExplainer method), 543
tuple_function() (in module sage.misc.latex), 483
tuple_function() (in module sage.misc.mathml), 457
type (sage.misc.package.PackageInfo attribute), 596
typecall() (in module sage.misc.classcall_metaclass), 113

U
UNICODE() (sage.misc.explain_pickle.PickleExplainer method), 543
unicode_art() (in module sage.typeset.unicode_art), 402
unicode_subscript() (in module sage.typeset.unicode_art), 404
unicode_superscript() (in module sage.typeset.unicode_art), 404
UnicodeArt (class in sage.typeset.unicode_art), 402
uniform() (in module sage.misc.prandom), 85
union() (sage.databases.sql_db.SQLQuery method), 187
UnknownClass (class in sage.misc.unknown), 88
UnknownError, 88
unpickle_all() (in module sage.misc.persist), 534
unpickle_append() (in module sage.misc.explain_pickle), 553
unpickle_build() (in module sage.misc.explain_pickle), 553
unpickle_extension() (in module sage.misc.explain_pickle), 554
unpickle_function() (in module sage.misc.fpickle), 538
unpickle_global() (in module sage.misc.persist), 535
unpickle_instantiate() (in module sage.misc.explain_pickle), 554
unpickle_newobj() (in module sage.misc.explain_pickle), 554
unpickle_persistent() (in module sage.misc.explain_pickle), 555
unpickle_method() (in module sage.misc.fpickle), 538
unpickleModule() (in module sage.misc.fpickle), 538
unset_verbose_files() (in module sage.misc.verbose), 227
update() (sage.misc.converting_dict.KeyConvertingDict method), 124
use_variable() (sage.misc.sage_input.SageInputBuilder method), 433

V
vacuum() (sage.databases.sql_db.SQLDatabase method), 181
value() (sage.ext.fast_callable.ExpressionConstant method), 334
values() (sage.misc.cachefunc.FileCache method), 294
values() (sage.misc.shellcmd.WeakValueDictionary method), 320
values_list() (sage.misc.shellcmd.WeakValueDictionary method), 321
var() (sage.ext.fast_callable.ExpressionTreeBuilder method), 338
variable_index() (sage.ext.fast_callable.ExpressionVariable method), 339
variable_names() (in module sage.misc.defaults), 2
vector_delimiters() (sage.misc.latex.Latex method), 467
verbose() (in module sage.misc.verbose), 227
verify_column() (in module sage.databases.sql_db), 189
verify_operator() (in module sage.databases.sql_db), 190
verify_same() (in module sage.misc.sage_input), 439
verify_si_answer() (in module sage.misc.sage_input), 440
verify_type() (in module sage.databases.sql_db), 190
version() (in module sage.misc.banner), 571
version_dict() (in module sage.misc.banner), 571
view() (in module sage.misc.latex), 484
Viewer (class in sage.misc.viewer), 575
visit_arg() (sage.misc.sageinspect.SageArgSpecVisitor method), 650
visit_BinOp() (sage.misc.sageinspect.SageArgSpecVisitor method), 645
visit_BoolOp() (sage.misc.sageinspect.SageArgSpecVisitor method), 646
visit_Compare() (sage.misc.sageinspect.SageArgSpecVisitor method), 646
visit_Dict() (sage.misc.sageinspect.SageArgSpecVisitor method), 647
visit_List() (sage.misc.sageinspect.SageArgSpecVisitor method), 647
visit_Name() (sage.misc.sageinspect.SageArgSpecVisitor method), 648
visit_NameConstant() (sage.misc.sageinspect.SageArgSpecVisitor method), 648
visit_Num() (sage.misc.sageinspect.SageArgSpecVisitor method), 649
visit_Str() (sage.misc.sageinspect.SageArgSpecVisitor method), 649
visit_Tuple() (sage.misc.sageinspect.SageArgSpecVisitor method), 649
visit_UnaryOp() (sage.misc.sageinspect.SageArgSpecVisitor method), 650
vonmisesvariate() (in module sage.misc.prandom), 85

W
warning() (in module sage.misc.superseded), 196
weak_cached_function() (in module sage.misc.cachefunc), 304
WeakCachedFunction (class in sage.misc.cachefunc), 295
WeakValueDictEraser (class in sage.misc.weak_dict), 310
WeakValueDictionary (class in sage.misc.weak_dict), 311
weibullvariate() (in module sage.misc.prandom), 86
width() (sage.typeset.character_art.CharacterArt method), 391
WithEqualityById (class in sage.misc.fast_methods), 117
word_wrap() (in module sage.misc.misc), 213
Wrapper (class in sage.ext.fast_callable), 347

X
xinterval() (in module sage.misc.functional), 62
xmrange (class in sage.misc.mrange), 131
xmrange_iter (class in sage.misc.mrange), 133
xsrange() (in module sage.arith.srange), 142

Z
ZZ_seed() (sage.misc.randstate.randstate method), 73