
Release 9.4

The Sage Development Team

Aug 24, 2021
## CONTENTS

1. General Infrastructure .................................................. 1
2. Programming Utilities .................................................. 49
3. Code Evaluation .......................................................... 229
4. Formatted Output ......................................................... 247
5. Saving and Loading Sage Objects ................................... 315
6. Interactive Usage Support .............................................. 349
7. Development Tools ...................................................... 387
8. Low-Level Utilities ....................................................... 441
9. Indices and Tables ....................................................... 459

Python Module Index ...................................................... 461

Index .............................................................................. 463
1.1 Default Settings

AUTHORS: William Stein and David Kohel

\texttt{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:

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

EXAMPLES:

\begin{verbatim}
\texttt{sage: from sage.misc.defaults import latex_variable_names
sage: latex_variable_names(0)
()}
\texttt{sage: latex_variable_names(1, 'a')
('a',)}
\texttt{sage: latex_variable_names(3, beta)
('beta_{0}', 'beta_{1}', 'beta_{2}')}
\texttt{sage: latex_variable_names(3, r'beta')
('\beta_{0}', '\beta_{1}', '\beta_{2}')}
\end{verbatim}

\texttt{sage.misc.defaults.series_precision()}

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

EXAMPLES:

\begin{verbatim}
\texttt{sage: series_precision()}
20
\end{verbatim}

\texttt{sage.misc.defaults.set_default_variable_name(name, separator=")

Change the default variable name and separator.

\texttt{sage.misc.defaults.set_series_precision(prec)}

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

EXAMPLES:

\begin{verbatim}
\texttt{sage: set_series_precision(5)
\texttt{sage: series_precision()}
\end{verbatim}
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
- names 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 foo(x) instead of x.foo() in certain common cases.

**AUTHORS:**
- William Stein: Initial version
- David Joyner (2005-12-20): More Examples

sage.misc.functional.N(x, prec=None, digits=None, algorithm=None)
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:**
You can also usually use method notation:

```
sage: (pi^2 + e).n()
12.5878862295484
sage: (pi^2 + e).numerical_approx()
12.5878862295484
```

Vectors and matrices may also have their entries approximated:

```
sage: v = vector(RDF, [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: v = vector(CDF, [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: v.parent()
Vector space of dimension 3 over Complex Field with 53 bits of precision
sage: v.n(prec=20)
(1.0000, 2.0000, 3.0000)
sage: u = vector(QQ, [1/2, 1/3, 1/4])
sage: n(u, prec=15)
(0.5000, 0.3333, 0.2500)
sage: n(u, digits=5)
(0.50000, 0.33333, 0.25000)
```

```
sage: v = vector(QQ, [1/2, 0, 0, 1/3, 0, 0, 0, 1/4], sparse=True)
sage: v.numerical_approx(digits=4)
sage: u.is_sparse()
True
sage: u
```

(continues on next page)
sage: A = matrix(QQ, 2, 3, range(6))
sage: A.n()
\[
\begin{bmatrix}
0.000000000000000 & 1.00000000000000 & 2.00000000000000 \\
3.00000000000000 & 4.00000000000000 & 5.00000000000000 \\
\end{bmatrix}
\]

sage: B = matrix(Integers(12), 3, 8, srange(24))
sage: N(B, digits=2)
\[
\begin{bmatrix}
0.00 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\
8.0 & 9.0 & 10. & 11. & 0.00 & 1.0 & 2.0 & 3.0 \\
4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0 & 10. & 11. \\
\end{bmatrix}
\]

Internally, numerical approximations of real numbers are stored in base-2. Therefore, numbers which look the same in their decimal expansion might be different:

sage: x=N(pi, digits=3); x
3.14
sage: y=N(3.14, digits=3); y
3.14
sage: x==y
False
sage: x.str(base=2)
'11.001001000100'
sage: y.str(base=2)
'11.001000111101'

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

sage: CC(-5).n(prec=100)
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 100 bits, use at most 53 bits
sage: n(1.3r, digits=20)
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 70 bits, use at most 53 bits
sage: RealField(24).pi().n()
Traceback (most recent call last):
...
TypeError: cannot approximate to a precision of 53 bits, use at most 24 bits

As an exceptional case, digits=1 usually leads to 2 digits (one significant) in the decimal output (see trac ticket #11647):

sage: N(pi, digits=1)
3.2
sage: N(pi, digits=2)
3.1
sage: N(100*pi, digits=1)
320.
sage: N(100*pi, digits=2)
310.
In the following example, \( \pi \) and 3 are both approximated to two bits of precision and then subtracted, which kills two bits of precision:

\begin{verbatim}
    sage: N(pi, prec=2)
    3.0
    sage: N(3, prec=2)
    3.0
    sage: N(pi - 3, prec=2)
    0.00
\end{verbatim}

`sage.misc.functional.additive_order(x)`
Return the additive order of \( x \).

**EXAMPLES:**

\begin{verbatim}
    sage: additive_order(5)
    +Infinity
    sage: additive_order(Mod(5,11))
    11
    sage: additive_order(Mod(4,12))
    3
\end{verbatim}

`sage.misc.functional.base_field(x)`
Return the base field over which \( x \) is defined.

**EXAMPLES:**

\begin{verbatim}
    sage: R = PolynomialRing(GF(7), 'x')
    sage: base_ring(R)
    Finite Field of size 7
    sage: base_field(R)
    Finite Field of size 7
\end{verbatim}

This catches base rings which are fields as well, but does not implement a `base_field` method for objects which do not have one:

\begin{verbatim}
    sage: R.base_field()
    Traceback (most recent call last):
    ...  
    AttributeError: 'PolynomialRing_dense_mod_p_with_category' object has no attribute 'base_field'
\end{verbatim}

`sage.misc.functional.base_ring(x)`
Return the base ring over which \( x \) is defined.

**EXAMPLES:**

\begin{verbatim}
    sage: R = PolynomialRing(GF(7), 'x')
    sage: base_ring(R)
    Finite Field of size 7
\end{verbatim}

`sage.misc.functional.basis(x)`
Return the fixed basis of \( x \).

**EXAMPLES:**

1.2. Functional notation
sage: V = VectorSpace(QQ,3)
sage: S = V.subspace([[1,2,0],[2,2,-1]])
sage: basis(S)
[(1, 0, -1),
 (0, 1, 1/2)]

sage.misc.functional.category(x)
Return the category of $x$.

EXAMPLES:

sage: V = VectorSpace(QQ,3)
sage: category(V)
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:

sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: charpoly(A)  
x^3 - 15*x^2 - 18*x
sage: charpoly(A, 't')  
t^3 - 15*t^2 - 18*t
sage: k.<alpha> = GF(7^10); k
Finite Field in alpha of size 7^10
sage: alpha.charpoly('T')  
T^10 + T^6 + T^5 + 4*T^4 + T^3 + 2*T^2 + 3*T + 3
sage: characteristic_polynomial(alpha, 'T')  
T^10 + T^6 + T^5 + 4*T^4 + T^3 + 2*T^2 + 3*T + 3

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

sage: y = var('y')
sage: a = matrix([[x,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
sage: characteristic_polynomial(a).list()
[x, -3*x - 1, 3*x + 3, -x - 3, 1]
sage: b = matrix([[y^(1/2),0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
sage: charpoly(b).list()
[sqrt(y), -3*sqrt(y) - 1, 3*sqrt(y) + 3, -sqrt(y) - 3, 1]

sage.misc.functional.charpoly(x, var='x')
Return the characteristic polynomial of $x$ in the given variable.

EXAMPLES:
sage: charpoly(A)
x^3 - 15*x^2 - 18*x
sage: charpoly(A, 't')
t^3 - 15*t^2 - 18*t

sage: k.<alpha> = GF(7^10); k
Finite Field in alpha of size 7^10
sage: alpha.charpoly('T')
T^10 + T^6 + T^5 + 4*T^4 + T^3 + 2*T^2 + 3*T + 3

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 (trac ticket #14403):

sage: y = var('y')

sage: a = matrix([[x,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
sage: characteristic_polynomial(a).list()  
[x, -3*x - 1, 3*x + 3, -x - 3, 1]

sage: b = matrix([[y^(1/2),0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
sage: charpoly(b).list()  
[sqrt(y), -3*sqrt(y) - 1, 3*sqrt(y) + 3, -sqrt(y) - 3, 1]

sage.misc.functional.coerce(P, x)
Coerce x to type P if possible.

EXAMPLES:

sage: type(5)  
<type 'sage.rings.integer.Integer'>

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

sage.misc.functional.cyclotomic_polynomial(n, var='x')
Return the n\text{th} cyclotomic polynomial.

EXAMPLES:

sage: cyclotomic_polynomial(3)  
x^2 + x + 1

sage: cyclotomic_polynomial(4)  
x^2 + 1

sage: cyclotomic_polynomial(9)  
x^6 + x^3 + 1

sage: cyclotomic_polynomial(10)  
x^4 - x^3 + x^2 - x + 1

sage: cyclotomic_polynomial(11)  
x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1

sage.misc.functional.decomposition(x)
Return the decomposition of x.

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

sage: G.<a,b> = DirichletGroup(20)
sage: c = a*b
dsage: 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

sage.misc.functional.denominator(x)
Return the denominator of x.

EXAMPLES:

sage: denominator(17/11111)
11111
sage: R.<x> = PolynomialRing(QQ)
sage: F = FractionField(R)
sage: r = (x+1)/(x-1)
sage: denominator(r)
x - 1

sage.misc.functional.det(x)
Return the determinant of x.

EXAMPLES:

sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: det(A)
0

sage.misc.functional.dim(x)
Return the dimension of x.

EXAMPLES:

sage: V = VectorSpace(QQ,3)
sage: S = V.subspace([[1,2,0],[2,2,-1]])
sage: dimension(S)
2

sage.misc.functional.dimension(x)
Return the dimension of x.

EXAMPLES:

sage: V = VectorSpace(QQ,3)
sage: S = V.subspace([[1,2,0],[2,2,-1]])
sage: dimension(S)
2

sage.misc.functional.disc(x)
Return the discriminant of x.

EXAMPLES:

sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^29 - 17*x - 1, 'alpha')
sage: K = S.number_field()
sage: discriminant(K)
-15975100446626038280218213241591829458737190477345113376757479850566957249523

sage.misc.functional.discriminant(x)
Return the discriminant of x.

EXAMPLES:

sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^29 - 17*x - 1, 'alpha')
sage: K = S.number_field()
sage: discriminant(K)
-15975100446626038280218213241591829458737190477345113376757479850566957249523

sage.misc.functional.eta(x)
Return the value of the \( \eta \) function at x, which must be in the upper half plane.

The \( \eta \) function is

\[
\eta(z) = e^{\pi i z / 12} \prod_{n=1}^{\infty} (1 - e^{2\pi i n z})
\]

EXAMPLES:

sage: eta(1+I)
0.7420487758365647 + 0.1988313702299107*I

sage.misc.functional.fcp(x, var='x')
Return the factorization of the characteristic polynomial of x.

EXAMPLES:

sage: M = MatrixSpace(QQ,3,3)
sage: A = M([[1,2,3,4,5,6,7,8,9]])
sage: fcp(A, 'x')
x * (x^8 - 15*x - 18)

sage.misc.functional.gen(x)
Return the generator of x.

EXAMPLES:

sage: R.<x> = QQ[]; R
Univariate Polynomial Ring in x over Rational Field
sage: gen(R)
sage.misc.functional.gens(x)
Return the generators of x.

EXAMPLES:

```sage
def g(x):
    return x
```

sage.misc.functional.hecke_operator(x, n)
Return the \( n \)-th Hecke operator \( T_n \) acting on \( x \).

EXAMPLES:

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

sage.misc.functional.image(x)
Return the image of \( x \).

EXAMPLES:

```sage
M = MatrixSpace(QQ,3,3)
sage: A = M([[1,2,3],[4,5,6],[7,8,9]])
sage: image(A)
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1 0 -1]
[ 0 1 2]
```

sage.misc.functional.integral(x, *args, **kwds)
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:
sage: f = cyclotomic_polynomial(10)
sage: integral(f)
1/5*x^5 - 1/4*x^4 + 1/3*x^3 - 1/2*x^2 + x

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

sage: y = var('y')
sage: integral(sin(x),y)
y*sin(x)

sage: integral(sin(x), x, 0, pi/2)
1
sage: integral(sin(x), x, 0, pi/2)
1
sage: integral(exp(-x), (x, 1, oo))
e^(-1)

Numerical approximation:

sage: h = integral(tan(x)/x, (x, 1, pi/3)); h
integrate(tan(x)/x, x, 1, 1/3*pi)
sage: h.n()
0.07571599101...

Specific algorithm can be used for integration:

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

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

EXAMPLES:

sage: integral_closure(QQ)
Rational Field
sage: K.<a> = QuadraticField(5)
sage: O2 = K.order(2*a); O2
Order in Number Field in a with defining polynomial x^2 - 5 with a = 2.
˓→236067977499790?
sage: integral_closure(O2)
Maximal Order in Number Field in a with defining polynomial x^2 - 5 with a = 2.
˓→236067977499790?

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.

1.2. Functional notation 11
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
sage: integral(sin(x),x)
-cos(x)
```

```python
sage: y = var('y')
sage: integral(sin(x),y)
y*sin(x)
```

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

Numerical approximation:

```python
sage: h = integral(tan(x)/x, (x, 1, pi/3)); h
integrate(tan(x)/x, x, 1, 1/3*pi)
sage: h.n()
0.07571599101...
```

Specific algorithm can be used for integration:

```python
sage: integral(sin(x)^2, x, algorithm='maxima')
1/2*x - 1/4*sin(2*x)
sage: integral(sin(x)^2, x, algorithm='sympy')
-1/2*cos(x)*sin(x) + 1/2*x
```

**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
```

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

Return whether or not x is commutative.

**EXAMPLES:**

```python
sage: R = PolynomialRing(QQ, 'x')
sage: is_commutative(R)
True
```
sage.misc.functional.is_even(x)
Return whether or not an integer x is even, e.g., divisible by 2.

EXAMPLES:

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

sage.misc.functional.is_field(x, proof=True)
Return whether or not x is a field.
Alternatively, one can use x in Fields().

EXAMPLES:

```
sage: R = PolynomialRing(QQ, 'x')
sage: F = FractionField(R)
sage: is_field(F)
True
```

sage.misc.functional.is_integrally_closed(x)
Return whether x is integrally closed.

EXAMPLES:

```
sage: is_integrally_closed(QQ)
True
sage: K.<a> = NumberField(x^2 + 189*x + 394)
sage: R = K.order(2*a)
sage: is_integrally_closed(R)
False
```

sage.misc.functional.is_odd(x)
Return whether or not x is odd.
This is by definition the complement of is_even().

EXAMPLES:

```
sage: is_odd(-2)
False
sage: is_odd(-3)
True
sage: is_odd(0)
False
sage: is_odd(1)
True
```

sage.misc.functional.isqrt(x)
Return an integer square root, i.e., the floor of a square root.

EXAMPLES:
sage: isqrt(10)
3
sage: isqrt(10r)
3

sage.misc.functional.kernel(x)
Return the left kernel of x.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,2)
sage: A = M([1,2,3,4,5,6])
sage: kernel(A)
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2 1]
sage: kernel(A.transpose())
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

Here are two corner cases:

```
sage: M = MatrixSpace(QQ,0,3)
sage: A = M([])
sage: kernel(A)
Vector space of degree 0 and dimension 0 over Rational Field
Basis matrix:
[]
sage: kernel(A.transpose()).basis()
[(1, 0, 0),
 (0, 1, 0),
 (0, 0, 1)]
```

sage.misc.functional.krull_dimension(x)
Return the Krull dimension of x.

EXAMPLES:

```
sage: krull_dimension(QQ)
0
sage: krull_dimension(ZZ)
1
sage: krull_dimension(ZZ[sqrt(5)])
1
sage: U.<x,y,z> = PolynomialRing(ZZ,3); U
Multivariate Polynomial Ring in x, y, z over Integer Ring
sage: U.krull_dimension()
4
```

sage.misc.functional.lift(x)
Lift an object of a quotient ring $R/I$ to $R$.

EXAMPLES:
We lift an integer modulo 3:

```
sage: Mod(2,3).lift()
sage: 2
```

We lift an element of a quotient polynomial ring:

```
sage: R.<x> = QQ['x']
sage: S.<xmod> = R.quo(x^2 + 1)
sage: lift(xmod-7)
sage: x - 7
```

`sage.misc.functional.log(x, b=None)`

Return the log of \(x\) to the base \(b\). The default base is \(e\).

DEPRECATED by trac ticket #19444

**INPUT:**
- \(x\) – number
- \(b\) – base (default: None, which means natural log)

**OUTPUT:** number

**Note:** In Magma, the order of arguments is reversed from in Sage, i.e., the base is given first. We use the opposite ordering, so the base can be viewed as an optional second argument.

**EXAMPLES:**

```
sage: from sage.misc.functional import log
sage: log(e^2)
```

```
doctest:warning...
DeprecationWarning: use .log() or log() from sage.functions.log instead
See http://trac.sagemath.org/19444 for details.
```

```
sage: log(16,2)
sage: 4
```

```
sage: log(3.)
sage: 1.09861228866811
```

```
sage: log(float(3))  # abs tol 1e-15
```

```
sage: 1.0986122886681098
```

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

Return the minimal polynomial of \(x\).

**EXAMPLES:**

```
sage: a = matrix(ZZ, 2, [1..4])
sage: minpoly(a)
```

```
x^2 - 5*x - 2
```

```
sage: minpoly(a, 't')
```

```
t^2 - 5*t - 2
```

```
sage: minimal_polynomial(a)
```

```
x^2 - 5*x - 2
```

(continues on next page)
sage: minimal_polynomial(a, 'theta')
theta^2 - 5*theta - 2

sage.misc.functional.minpoly(x, var='x')
Return the minimal polynomial of x.

EXAMPLES:

sage: a = matrix(ZZ, 2, [1..4])
sage: minpoly(a)
x^2 - 5*x - 2
sage: minpoly(a, 't')
t^2 - 5*t - 2
sage: minimal_polynomial(a)
x^2 - 5*x - 2
sage: minimal_polynomial(a, 'theta')
theta^2 - 5*theta - 2

sage.misc.functional.multiplicative_order(x)
Return the multiplicative order of x, if x is a unit, or raise ArithmeticError otherwise.

EXAMPLES:

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

sage.misc.functional.n(x, prec=None, digits=None, algorithm=None)
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:
You can also usually use method notation:

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

Vectors and matrices may also have their entries approximated:

sage: v = vector(RDF, [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: v = vector(CDF, [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: v.parent()  # Vector space of dimension 3 over Complex Field with 53 bits of precision
sage: v.n(prec=20)
(1.0000, 2.0000, 3.0000)

sage: u = vector(QQ, [1/2, 1/3, 1/4])
sage: n(u, prec=15)
(0.5000, 0.3333, 0.2500)
sage: n(u, digits=5)
(0.50000, 0.33333, 0.25000)

sage: v = vector(QQ, [1/2, 0, 0, 1/3, 0, 0, 0, 1/4], sparse=True)
sage: u = v.numerical_approx(digits=4)
sage: u.is_sparse()
True
sage: u

(continues on next page)
Internally, numerical approximations of real numbers are stored in base-2. Therefore, numbers which look the same in their decimal expansion might be different:

```python
sage: x = N(pi, digits=3); x
3.14
sage: y = N(3.14, digits=3); y
3.14
sage: x == y
False
sage: x.str(base=2)
'11.001001000100'
sage: y.str(base=2)
'11.001000111101'
```

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

```python
sage: CC(-5).n(prec=100)
Traceback (most recent call last):
  ... TypeError: cannot approximate to a precision of 100 bits, use at most 53 bits
sage: n(1.3r, digits=20)
Traceback (most recent call last):
  ... TypeError: cannot approximate to a precision of 70 bits, use at most 53 bits
sage: RealField(24).pi().n()
Traceback (most recent call last):
  ... TypeError: cannot approximate to a precision of 53 bits, use at most 24 bits
```

As an exceptional case, `digits=1` usually leads to 2 digits (one significant) in the decimal output (see trac ticket #11647):

```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:

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

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

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

**EXAMPLES:**

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

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

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()`

1.2. Functional notation

19
• `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: 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)
```

The norm of matrices:

```python
sage: z = 1 + 2*I
sage: norm(matrix([[z]]))
2.23606797749979
sage: M = matrix(ZZ, [[1,2,4,3], [-1,0,3,-10]])
```

```python
sage: norm(M) # abs tol 1e-14
10.690331129154467
sage: norm(CDF(z))
5.0
sage: norm(CC(z))
5.00000000000000
```

The norm of complex numbers:

```python
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:

```python
sage: a, b, c = var("a, b, c")
sage: assume((a, 'real'), (b, 'real'), (c, 'real'))
sage: z = a + b*I
sage: bool(norm(z).simplify() == a^2 + b^2)
True
sage: norm(a + b).simplify()
a^2 + 2*a*b + b^2
```

(continues on next page)
sage: v = vector([a, b, c])
 sage: bool(norm(v).simplify() == sqrt(a^2 + b^2 + c^2))
 True
 sage: forget()

sage.misc.functional.numerator(x)
Return the numerator of x.

EXAMPLES:

sage: R.<x> = PolynomialRing(QQ)
sage: F = FractionField(R)
sage: r = (x+1)/(x-1)
sage: numerator(r)
x + 1
 sage: numerator(17/11111)
17

sage.misc.functional.numerical_approx(x, prec=None, digits=None, algorithm=None)
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: numerical_approx(pi, 10)
3.1
 sage: numerical_approx(pi, digits=10)
3.141592654
 sage: numerical_approx(pi^2 + e, digits=20)
12.587886229548403854
 sage: n(pi^2 + e)
12.5878862295484
 sage: N(pi^2 + e)
12.5878862295484
 sage: n(pi^2 + e, digits=50)
12.587886229548403854194778471228813633070946500941
 sage: a = CC(-5).n(prec=40)
sage: b = ComplexField(40)\(-5\)
sage: a == b
True
 sage: parent(a) is parent(b)
True

\[\text{sage: numerical\_approx}(9)\]
\[9.00000000000000\]

You can also usually use method notation:

\[\text{sage: } (\pi^2 + e).n()\]
\[12.5878862295484\]
\[\text{sage: } (\pi^2 + e).\text{numerical\_approx()}\]
\[12.5878862295484\]

Vectors and matrices may also have their entries approximated:

\[\text{sage: } v = \text{vector}(\text{RDF}, \{1,2,3\})\]
\[\text{sage: } v.n()\]
\[(1.00000000000000, 2.00000000000000, 3.00000000000000)\]

\[\text{sage: } v = \text{vector}(\text{CDF}, \{1,2,3\})\]
\[\text{sage: } v.n()\]
\[(1.00000000000000, 2.00000000000000, 3.00000000000000)\]
\[\text{sage: } \_\text{.parent()}\]
Vector space of dimension 3 over Complex Field with 53 bits of precision
\[\text{sage: } v.n(\text{prec}=20)\]
\[(1.0000, 2.0000, 3.0000)\]

\[\text{sage: } u = \text{vector}(\text{QQ}, \{1/2, 1/3, 1/4\})\]
\[\text{sage: } n(u, \text{prec}=15)\]
\[(0.5000, 0.3333, 0.2500)\]
\[\text{sage: } n(u, \text{digits}=5)\]
\[(0.50000, 0.33333, 0.25000)\]

\[\text{sage: } v = \text{vector}(\text{QQ}, \{1/2, 0, 0, 1/3, 0, 0, 1/4\}, \text{sparse}=\text{True})\]
\[\text{sage: } u = v.\text{numerical\_approx}(\text{digits}=4)\]
\[\text{sage: } u.\text{is\_sparse()}\]
True
\[\text{sage: } u\]
\[(0.5000, 0.0000, 0.0000, 0.3333, 0.0000, 0.0000, 0.2500)\]

\[\text{sage: } A = \text{matrix}(\text{QQ}, 2, 3, \text{range}(6))\]
\[\text{sage: } A.n()\]
\[
\begin{bmatrix}
0.000000000000000 & 1.000000000000000 & 2.000000000000000 \\
3.000000000000000 & 4.000000000000000 & 5.000000000000000
\end{bmatrix}
\]

\[\text{sage: } B = \text{matrix}(\text{Integers}(12), 3, 8, \text{range}(24))\]
\[\text{sage: } N(B, \text{digits}=2)\]
\[
\begin{bmatrix}
0.00 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\
8.0 & 9.0 & 10. & 11. & 0.00 & 1.0 & 2.0 & 3.0 \\
4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0 & 10. & 11.
\end{bmatrix}
\]

Internally, numerical approximations of real numbers are stored in base-2. Therefore, numbers which look the same in their decimal expansion might be different:
Increasing the precision of a floating point number is not allowed:

```plaintext
sage: CC(-5).n(prec=100)
Traceback (most recent call last):
  ...TypeError: cannot approximate to a precision of 100 bits, use at most 53 bits
sage: n(1.3r, digits=20)
Traceback (most recent call last):
  ...TypeError: cannot approximate to a precision of 70 bits, use at most 53 bits
sage: RealField(24).pi().n()
Traceback (most recent call last):
  ...TypeError: cannot approximate to a precision of 53 bits, use at most 24 bits
```

As an exceptional case, digits=1 usually leads to 2 digits (one significant) in the decimal output (see trac ticket #11647):

```plaintext
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:

```plaintext
sage: N(pi, prec=2)
3.0
sage: N(3, prec=2)
3.0
sage: N(pi - 3, prec=2)
0.00
```

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

EXAMPLES:

```plaintext
sage: R, x = objgen(FractionField(QQ['x']))
sage: R
```

(continues on next page)
Fraction Field of Univariate Polynomial Ring in x over Rational Field
\texttt{sage: x}
x

\texttt{sage.misc.functional.objgens(x)}

\texttt{EXAMPLES:}

\begin{verbatim}
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)
\end{verbatim}

\texttt{sage.misc.functional.order(x)}

Return the order of \texttt{x}.

If \texttt{x} is a ring or module element, this is the additive order of \texttt{x}.

\texttt{EXAMPLES:}

\begin{verbatim}
sage: C = CyclicPermutationGroup(10)
sage: order(C)
10
sage: F = GF(7)
sage: order(F)
7
\end{verbatim}

\texttt{sage.misc.functional.quo(x, y, *args, **kwds)}

Return the quotient object \texttt{x/y}, e.g., a quotient of numbers or of a polynomial ring \texttt{x} by the ideal generated by \texttt{y}, etc.

\texttt{EXAMPLES:}

\begin{verbatim}
sage: quotient(5,6)
5/6
sage: quotient(5.,6.)
0.833333333333333
sage: R.<x> = ZZ[]; R
Univariate Polynomial Ring in x over Integer Ring
sage: I = Ideal(R, x^2+1)
sage: quotient(R, I)
Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 + 1
\end{verbatim}

\texttt{sage.misc.functional.quotient(x, y, *args, **kwds)}

Return the quotient object \texttt{x/y}, e.g., a quotient of numbers or of a polynomial ring \texttt{x} by the ideal generated by \texttt{y}, etc.

\texttt{EXAMPLES:}

\begin{verbatim}
sage: quotient(5,6)
5/6
sage: quotient(5.,6.)
0.833333333333333
sage: R.<x> = ZZ[]; R
Univariate Polynomial Ring in x over Integer Ring
\end{verbatim}
sage: I = Ideal(R, x^2+1)
sage: quotient(R, I)
Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 + 1

sage.misc.functional.rank(x)
Return the rank of x.

EXAMPLES:

We compute the rank of a matrix:

sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: rank(A)
2

We compute the rank of an elliptic curve:

sage: E = EllipticCurve([0,0,1,-1,0])
sage: rank(E)
1

sage.misc.functional.regulator(x)
Return the regulator of x.

EXAMPLES:

sage: regulator(NumberField(x^2-2,'a'))
0.881373587019543
sage: regulator(EllipticCurve('11a'))
1.00000000000000

sage.misc.functional.round(x, ndigits=0)
round(number[, ndigits]) - double-precision real number

Round a number to a given precision in decimal digits (default 0 digits). If no precision is specified this just calls
the element’s .round() method.

EXAMPLES:

sage: round(sqrt(2),2)
1.41
sage: q = round(sqrt(2),5); q
1.41421
sage: type(q)
<type 'sage.rings.real_double.RealDoubleElement'>
sage: q = round(sqrt(2)); q
1
sage: type(q)
<type 'sage.rings.integer.Integer'>
sage: round(pi)
3
sage: b = 5.4999999999999999
sage: round(b)
5

1.2. Functional notation
This example addresses trac ticket #23502:

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

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

```
sage: round(sqrt(Integer('1'*1000)),2)
+infinity
```

IMPLEMENTATION: If ndigits is specified, it calls Python’s builtin round function, and converts the result to a real double field element. Otherwise, it tries the argument’s round() method; if that fails, it reverts to the builtin round function, converted to a real double field element.

**Note:** This is currently slower than the builtin round function, since it does more work - i.e., allocating an RDF element and initializing it. To access the builtin version do `import builtins; builtins.round`.

---

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

Return the square free part of $x$, i.e., a divisor $z$ such that $x = zy^2$, for a perfect square $y^2$.

**EXAMPLES:**

```
sage: squarefree_part(100)
1
sage: squarefree_part(12)
3
sage: squarefree_part(10)
10
sage: squarefree_part(216r) # see #8976
6
```

```
sage: x = QQ['x'].0
sage: S = squarefree_part(-9*x*(x-6)^7*(x-3)^2); S
-9*x^2 + 54*x
sage: S.factor()
(-9) * (x - 6) * x
```

```
sage: f = (x^3 + x + 1)^3*(x-1); f
x^10 - x^9 + 3*x^8 + 3*x^5 - 2*x^4 - x^3 - 2*x - 1
sage: g = squarefree_part(f); g
x^4 - x^3 + x^2 - 1
sage: g.factor()
(x - 1) * (x^3 + x + 1)
```

---

`sage.misc.functional.symbolic_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: 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)
```

```
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:

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

1.2. Functional notation
sage: sum(1/k^4, k, 1, oo)
1/90*pi^4

sage: sum(1/k^5, k, 1, oo)
zeta(5)

Warning: This function only works with symbolic expressions. To sum any other objects like list elements or function return values, please use python summation, see http://docs.python.org/library/functions.html#sum

In particular, this does not work:

sage: n = var('n')
sage: mylist = [1,2,3,4,5]
sage: sum(mylist[n], n, 0, 3)
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

Also, only a limited number of functions are recognized in symbolic sums:

sage: sum(valuation(n,2),n,1,5)
Traceback (most recent call last):
  ...TypeError: unable to convert n to an integer

Again, use python sum():

sage: sum(valuation(n+1,2) for n in range(5))
3

(Now back to the Sage sum examples)

A well known binomial identity:

sage: sum(binomial(n,k), k, 0, n)
2^n

The binomial theorem:

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

sage: sum(k * binomial(n, k), k, 1, n)
2^(n - 1)*n

sage: sum((-1)^k*binomial(n,k), k, 0, n)
0
Another binomial identity (trac ticket #7952):

```python
sage: t,k,i = var('t,k,i')
sage: sum(binomial(i+t,t),i,0,k)
binomial(k + t + 1, t + 1)
```

Summing a hypergeometric term:

```python
sage: sum(binomial(n,k) * factorial(k) / factorial(n+1+k), k, 0, n)
1/2*sqrt(pi)/factorial(n + 1/2)
```

We check a well known identity:

```python
sage: bool(sum(k^3, k, 1, n) == sum(k, k, 1, n)^2)
True
```

A geometric sum:

```python
sage: a, q = var('a, q')
sage: sum(a*q^k, k, 0, n)
(a*q^(n + 1) - a)/(q - 1)
```

The geometric series:

```python
sage: assume(abs(q) < 1)
sage: sum(a*q^k, k, 0, oo)
-a/(q - 1)
```

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

```python
sage: forget()
sage: assume(q > 1)
sage: sum(a*q^k, k, 0, oo)
Traceback (most recent call last):
... ValueError: Sum is divergent.
```

This summation only Mathematica can perform:

```python
sage: sum(1/(1+k^2), k, -oo, oo, algorithm = 'mathematica')  # optional - mathematica
-\pi*coth(\pi)
```

Use Maple as a backend for summation:

```python
sage: sum(binomial(n,k)*x^k, k, 0, n, algorithm = 'maple')  # optional - maple
(x + 1)^n
```

Python ints should work as limits of summation (trac ticket #9393):

```python
sage: sum(x, x, 1r, 5r)
15
```
Note:

1. Sage can currently only understand a subset of the output of Maxima, Maple and Mathematica, so even if the chosen backend can perform the summation the result might not be convertible into a Sage expression.

sage.misc.functional.transpose(x)

Return the transpose of x.

EXAMPLES:

```python
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: transpose(A)

[1 4 7]
[2 5 8]
[3 6 9]
```

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

Iterator over the integers between a and b, inclusive.

EXAMPLES:

```python
sage: I = xinterval(2,5); I  # py2
xrange(2, 6)
sage: I = xinterval(2,5); I  # py3
range(2, 6)
sage: 5 in I
True
sage: 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.

```sage
sage: K.<x> = QQ[]
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: rgp = Gp()
sage: def gap_randstring(n):
....:     current_randstate().set_seed_gap()
....:     return gap(n).SCRRandomString()
sage: def rtest():
....:     current_randstate().set_seed_gp(rgp)
....:     return (ZZ.random_element(1000), RR.random_element(),
....:            K.random_element(), G.random_element(),
....:            gap_randstring(5),
....:            rgp.random(), ntl.ZZ_random(99999),
....:            random())
```

The above test shows the results of six different random number generators, in three different processes. The random elements from `ZZ`, `RR`, and `K` all derive from a single GMP-based random number generator. The random element from `G` comes from a GAP subprocess. The random “string” (5-element binary list) is also from a GAP subprocess, using the “classical” GAP random generator. The random number from `rgp` is from a Pari/gp subprocess. NTL's `ZZ_random` uses a separate NTL random number generator in the main Sage process. And `random()` is from a Python `random.Random` object.

Here we see that setting the random number seed really does make the results of these random number generators reproducible.

```sage
sage: set_random_seed(0)
sage: print(rtest())
(303, -0.266166246380421, 1/6, (1,2), [ 0, 1, 1, 0, 0 ], 265625921, 79302, 0.
   → 2450652680687958)
sage: set_random_seed(1)
sage: print(rtest())
(978, 0.0557699430711638, -1/8*x^2 - 1/2*x + 1/2, (1,2,3), [ 1, 0, 0, 0, 1 ], 807447831,
   → 23865, 0.6170498912488264)
sage: set_random_seed(2)
sage: print(rtest())
(207, -0.0141049486533456, (1,3)(4,5), [ 1, 0, 1, 1, 1 ], 1642898426, 16190, 0.
   → 934331114872127)
```

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.)

```python
sage: set_random_seed(12345)
sage: initial_seed()
12345L
sage: print(rtest())
(720, -0.612180244315804, 0, (1,3), [ 1, 0, 1, 1, 0 ], 1911581957, 65175, 0.
˓→804382951758298)
sage: initial_seed()
12345L
```

If `set_random_seed()` is called with no arguments, then a new seed is automatically selected. On operating systems that support it, the new seed comes from `os.urandom()`; this is intended to be a truly random (not pseudo-random), cryptographically secure number. (Whether it is actually cryptographically secure depends on operating system details that are outside the control of Sage.)

If `os.urandom()` is not supported, then the new seed comes from the current time, which is definitely not cryptographically secure.

```python
sage: set_random_seed()
sage: r = rtest()
sage: r
# random
(989, -0.407373370020575, 6/7*x^2 + 1, (1,2,3)(4,5), 985329107, 21461, 0.
˓→30047071049504859)
```

After setting a new random number seed with `set_random_seed()`, we can use `initial_seed()` to see what seed was automatically selected, and call `set_random_seed()` to restart the same random number sequence.

```python
sage: s = initial_seed()
sage: s
# random
336237747258024892839280045662L
sage: set_random_seed(s)
sage: r2 = rtest()
sage: r == r2
True
```

Whenever Sage starts, `set_random_seed()` is called just before command line interaction starts; so every Sage run starts with a different random number seed. This seed can be recovered with `initial_seed()` (as long as the user has not set a different seed with `set_random_seed()`), so that the results of this run can be reproduced in another run; or this automatically selected seed can be overridden with, for instance, `set_random_seed(0)`.

We can demonstrate this startup behavior by running a new instance of Sage as a subprocess.

```python
sage: subsage = Sage()
sage: s = ZZ(subsage('initial_seed()'))
sage: r = ZZ(subsage('ZZ.random_element(2^200)'))
sage: s
# random
161165040149656168853863459174502758403
sage: r
# random
1273828861620427462924151488498075119241254209468761367941442
sage: set_random_seed(s)
sage: r == ZZ.random_element(2^200)
True
```

Note that wrappers of all the random number generation methods from Python’s `random` module are available at the Sage command line, and these wrappers are properly affected by `set_random_seed()`.
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.

```plaintext
These computations use pseudo-random numbers, so we set the seed for reproducible testing.
```

```python
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`.

```python
i = 0
for _ in range(10):
    print(i, random(), getrandbits(20), uniform(5.0, 10.0), normalvariate(0, 1))
    i += 1
```
We get slightly different results with an intervening \texttt{with seed}.

We can see that \texttt{r2} and \texttt{r2m} are the same except for the call to \texttt{ntl.ZZ_random()}, which produces different results with and without the \texttt{with seed}.

However, we do still get a partial form of isolation, even in this case, as we see in this example:

The NTL results after the \texttt{with seed} don’t depend on how many NTL random numbers were generated inside the \texttt{with seed}.

(In general, the above code is not exactly equivalent to the \texttt{with} statement, because if an exception happens in the body, the real \texttt{with} statement will pass the exception information as parameters to the \texttt{__exit__} method. However, our \texttt{__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 methods on parents), you don’t need to do anything special; the other code presumably already interacts with this module correctly. Otherwise, it depends on what random number generator you want to use.

- **gmp_randstate_t** – If you want to use some random number generator that takes a gmp_randstate_t (like mpz_urandomm or mpfr_urandomb), then use code like the following:

```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).

- **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:

```python
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
```

`sage.misc.randstate.benchmark_mt()`
This function was used to test whether moving from libc to GMP’s Mersenne Twister for random numbers would be a significant slowdown.

EXAMPLES:

```
sage: from sage.misc.randstate import benchmark_libc, benchmark_mt
sage: timeit('benchmark_libc()')  # random
125 loops, best of 3: 1.95 ms per loop
sage: timeit('benchmark_mt()')  # random
125 loops, best of 3: 2.11 ms per loop
```

`sage.misc.randstate.current_randstate()`
Return the current random number state.

EXAMPLES:

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

`sage.misc.randstate.initial_seed()`
Returns the initial seed used to create the current `randstate`.

EXAMPLES:

```
sage: set_random_seed(42)
sage: initial_seed()
42L
```
If you set a random seed (by failing to specify the seed), this is how you retrieve the seed actually chosen by Sage. This can also be used to retrieve the seed chosen for a new Sage run (if the user has not used `set_random_seed()`).

```
sage: set_random_seed()  # random
sage: initial_seed()  # random
121030915244661507561642968348336774L
```

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

Bases: object

The randstate class. This class keeps track of random number states and seeds. Type sage.misc.randstate? for much more information on random numbers in Sage.

ZZ_seed()

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

EXAMPLES:

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

long_seed()

Whencalledonthecurrent randstate, returns a 128-bit Python long suitable for seeding anothers random number generator.

EXAMPLES:
 sage: set_random_seed(1618)
sage: current_randstate().long_seed()
25605627977451409508607350947089272595L

**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:**

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

**seed()**

Return the initial seed of a *randstate* object. (This is not the current state; it does not change when you get random numbers.)

**EXAMPLES:**

```
sage: set_random_seed(0)
sage: from sage.misc.randstate import randstate
sage: r = randstate(314159)
sage: r.seed()
314159L
sage: r.python_random().random()
0.111439293741037
sage: r.seed()
314159L
```

**set_seed_gap()**

Checks to see if *self* was the most recent *randstate* to seed the GAP random number generator. If not, seeds the generator.

**EXAMPLES:**

```
sage: set_random_seed(99900000999)
sage: current_randstate().set_seed_gap()
sage: gap.Random(1, 10^50)
149673826332555434474532297768680634540939580077
sage: gap(35).SCRRandomString()
[ 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 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:**

```python
sage: set_random_seed(987654321)
sage: current_randstate().set_seed_gp()
sage: gp.random()
23289294
```

**set_seed_libc(force)**

Checks to see if `self` was the most recent `randstate` to seed the libc random number generator. If not, seeds the 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:**

```python
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:**

```python
sage: set_random_seed(2008)
```

This call is actually redundant; ntl.ZZ_random() will seed the generator itself. However, we put the call in to make the coverage tester happy.

```python
sage: current_randstate().set_seed_ntl(False)
sage: ntl.ZZ_random(10^40)
1495283511775355459459288047895196007
```

**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'
```
sage.misc.randstate.seed
   alias of sage.misc.randstate.randstate

sage.misc.randstate.set_random_seed(seed=\text{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:

\begin{Verbatim}
\sage: set_random_seed(5)
\sage: initial_seed()
5L
\end{Verbatim}

\section{1.4 Random Numbers with Python API}

\textbf{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.)

\begin{Verbatim}
\sage: from sage.misc.prandom import randrange
\sage: from sage.misc.randstate import current_randstate
\sage: def test1():
...    return sum([randrange(100) for \text{i in range(100)}])
\sage: def test2():
...    randrange = current_randstate().python_random().randrange
...    return sum([randrange(100) for \text{i in range(100)}])
\end{Verbatim}

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

\begin{Verbatim}
\sage: with seed(0): test1()
5169
\sage: with seed(0): test2()
5169
\sage: with seed(1): test1()
5097
\sage: with seed(1): test2()
\end{Verbatim}

\bigskip
\textit{continues on next page}
5097
sage: timeit('test1()') # random
625 loops, best of 3: 590 us per loop
sage: timeit('test2()') # random
625 loops, best of 3: 460 us per loop

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.

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:

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

sage.misc.prandom.choice(seq)
Choose a random element from a non-empty sequence.

EXAMPLES:

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)
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:

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

(continues on next page)
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

def 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

def sample = [gauss(0, 1) for i in range(3)]  # random
[0.9191011757657915, 0.7744526756246484, 0.8638996866800877]
sage: [gauss(0, 10) for i in range(3)]  # random
[24.91051749154448, -62.99272061579273, -8.199312536718...]
sage: [gauss(1000, 10) for i in range(3)]  # random
[998.759070045661, 996.1087338511692, 1010.1256817458031]

sage: getrandbits(k)
getrandbits(k) -> x. Generates a long int with k random bits.

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: lognormvariate(mu, sigma)
Log normal 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.

EXCEPTIONS:
sage: [lognormvariate(100, 10) for i in range(3)] # random
[2.941035568829246e+37, 2.2257548162070125e+38, 4.142299451717446e+43]

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, -1.170160636898928, 0.04324100555110143]
sage: [normalvariate(0, 100) for i in range(3)]
# random
[37.4595875941769, 159.634774323298, 124.102932124009]
sage: [normalvariate(1000, 10) for i in range(3)]
# random
[1008.5303090383741, 989.862492644895, 985.7728921150242]

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, 1.0155564009379579, 1.144232578983077]

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, 2, 0, 2, 2]
sage: all(t in [0, 1, 2] for t in s)
True
sage: -100 <= randint(-100, 10) <= 10
True

sage.misc.prandom.random()
Get the next random number in the range [0.0, 1.0).
EXAMPLES:
sage: sample = [random() for i in [1 .. 4]]; sample
# random
[0.11143923741037, 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:
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: s = randrange(0, 1000000, 1000)
sage: 0 <= s < 1000000
True
sage: s % 1000
0
sage: -100 <= randrange(-100, 10) < 10
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:

sage: 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

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:

```python
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:

```python
sage: s = uniform(0, 1); s  # random
0.111439293741037
sage: 0.0 <= s <= 1.0
True

sage: s = uniform(e, pi); s  # random
0.5143475134191677*pi + 0.4856524865808327*e
sage: bool(e <= s <= pi)
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:

```python
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
```

```
sage.misc.prandom.weibullvariate(alpha, beta)
```

Weibull distribution.

\(\alpha\) is the scale parameter and \(\beta\) is the shape parameter.

EXAMPLES:

```python
sage: sample = [weibullvariate(1, 3) for i in range(1, 5)]; sample  # random
[0.490697754634237, 0.897218556461213, 0.357573846531942, 0.739377255516847]
sage: all(s >= 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:

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

Using direct identification:
```

```python
sage: for n in [-3, 0, 12]:
    ....:     res = func(n)
    ....:     if res is True:
    ....:         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))
n=-3 is negative
n=0 is neither positive nor negative
n=12 is positive
```

Using `UnknownError`:

```python
sage: for n in [-3, 0, 12]:
    ....:     try:
    ....:         if func(n):
    ....:             print("n={} is positive".format(n))
```

(continues on next page)
.. code-block:: python

    n=-3 is negative
    n=0 is neither positive nor negative
    n=12 is positive

AUTHORS:


```python
sage.misc.unknown.Unknown = Unknown
```

class sage.misc.unknown.UnknownClass

Bases: sage.structure.unique_representation.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:**

```python
sage: 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__'

    is_optional()  
    Returns whether an abstract method is optional or not.

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

`sage.misc.abstract_method.abstract_method(f=None, optional=False)`  
Abstract methods
INPUT:

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

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

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

EXAMPLES:

We create a class with an abstract method:

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

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

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

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

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

(continues on next page)
sage: x = A()
sage: x.my_method
NotImplemented

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

```python
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.
```

**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:

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

Instead one needs to do:

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

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

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

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

**Todo:** look for similar decorators on the web, and merge

**Implementation details**

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

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

```python
sage: abstract_method(optional = True)  # py2
<function <lambda> at ...>
sage: abstract_method(optional = True)  # py3
<function abstract_method.<locals>.<lambda> at ...>
```
sage: abstract_method(optional = True)(banner)
<optional abstract method banner at ...>
sage: abstract_method(banner, optional = True)
<optional abstract method banner at ...>

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

Returns the required and optional abstract methods of the class

**EXAMPLES:**

```
sage: class AbstractClass:
    ....:     @abstract_method
    ....:     def required1(): pass
    ....:     @abstract_method(optional = True)
    ....:     def optional2(): pass
    ....:     @abstract_method(optional = True)
    ....:     def optional1(): pass
    ....:     @abstract_method
    ....:     def required2(): pass
sage: sage.misc.abstract_method.abstract_methods_of_class(AbstractClass)
{'optional': ['optional1', 'optional2'],
 'required': ['required1', 'required2']}
```

### 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 # just a workaround for Python_
    ....:     __misnaming nested classes
    ....:     class Inner:
    ....:         def __init__(self, *args):
    ....:             print(args)
    ....:         def f(self, *args):
    ....:             print("{} {}".format(self, args))
```

(continues on next page)
By default, when Inner is a class nested in Outer, accessing `outer.Inner` returns the Inner class as is:

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

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

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

This is similar to what happens with a static method:

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

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

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

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

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

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

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

Bindable Class gives this binding behavior to all its subclasses:

```
sage: from sage.misc.bindable_class importBindableClass
sage: class Outer:
....:     __metaclass__ = NestedClassMetaclass # just a workaround for Python
˓→misnaming nested classes
....:
....:     class Inner(BindableClass):
....:         " some documentation "
....:         def __init__(self, outer, *args):
....:             print("{} {}".format(outer, args))
```

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

```
sage: Outer.Inner
<class '__main__.Outer.Inner'>
```
However, outer.Inner(1,2,3) is equivalent to Outer.Inner(outer, 1,2,3):

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

**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'>]
```

Still, documentation works as usual:

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

```python
class sage.misc.bindable_class.BoundClass(*args)
    Bases: functools.partial
class sage.misc.bindable_class.Inner2
    Bases: sage.misc.bindable_classBindableClass
    Some documentation for Inner2
class sage.misc.bindable_class OUTER
    Bases: object
    A class with a bindable nested class, for testing purposes
class Inner
    Bases: sage.misc.bindable_classBindableClass
    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.
• Julian Rueth (2014-03-19): added decorator_keywords decorator

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

This function allows a decorator to have default arguments.

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

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

EXAMPLES:

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

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

A decorator for decorators with optional keyword arguments.

EXAMPLES:

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

This decorator can be called with and without arguments:

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

(continues on next page)
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
foo(1)
1

class sage.misc.decorators.infix_operator(precedence)
Bases: object
A decorator for functions which allows for a hack that makes the function behave like an infix operator.

This decorator exists as a convenience for interactive use.

EXAMPLES:

An infix dot product operator:

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

An infix element-wise addition operator:

sage: @infix_operator('add')
    def eadd(a, b):
        return a.parent([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:

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

class sage.misc.decorators.options(**options)
Bases: object
A decorator for functions which allows for default options to be set and reset by the end user. Additionally, if one needs to, one can get at the original keyword arguments passed into the decorator.

class sage.misc.decorators.rename_keyword(**options)
Bases: object
A decorator which renames keyword arguments and optionally deprecates the new keyword.

INPUT:

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

EXAMPLES:

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

To deprecate an old keyword:

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

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)
x^2
sage: g.__doc__
'My little function'
sage: from sage.misc.sageinspect import sage_getsource, sage_getsourcelines, sage_
    ....:     _getfile
```

(continues on next page)
Demonstrate that the argument description are retained from the decorated function through the special method (when left unchanged) (see trac ticket #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)
ArgSpec(args=['x'], varargs=None, keywords=None, defaults=None)
```

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

```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()
  "], ...)]
```

The `f` attribute of the decorated function refers to the original function:

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

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

```python
sage: def square_for_met(f):
    ...:     @sage_wraps(f)
    ...:     def new_f(self, x):
    ...:         return f(self,x)*f(self,x)
```

(continues on next page)
....: return new_f
sage: class T:
....: @square_for_met
....: def g(self, x):
....:     "My little method"
....:     return x
sage: t = T()
sage: t.g(2)
4
sage: t.g.__doc__
'My little method'

The bug described in trac ticket #11734 is fixed:

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

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!

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:

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

### 2.1.4 Constant functions

```python
class sage.misc.constant_function.ConstantFunction
Bases: sage.structure.sage_object.SageObject

A class for function objects implementing constant functions.

EXAMPLES:

```python
sage: f = ConstantFunction(3)
sage: f
The constant function (...) -> 3
sage: f()
3
sage: f(5)
3
```

Such a function could be implemented as a lambda expression, but this is not (currently) picklable:

```python
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: sage.misc.nested_class.NestedClassMetaclass

A metaclass providing support for special methods for classes.

From the Section Special method names of the Python Reference Manual:

‘a class cls can implement certain operations on its instances that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names’.

The purpose of this metaclass is to allow for the class cls to implement analogues of those special methods for the operations on the class itself.

Currently, the following special methods are supported:

• __classcall__ (and __classcall_private__) for customizing cls(...) (analogue of __call__).
• __classcontains__ for customizing membership testing x in cls (analogue of __contains__).
• __classget__ for customizing the binding behavior in foo.cls (analogue of __get__).

See the documentation of __call__() and of __get__() and __contains__() for the description of the respective protocols.

Warning: For technical reasons, __classcall__, __classcall_private__, __classcontains__, and __classget__ must be defined as staticmethod()’s, even though they receive the class itself as their first argument.

Warning: For efficiency reasons, the resolution for the special methods is done once for all, upon creation of the class. Thus, later dynamic changes to those methods are ignored. But see also _set_classcall().

ClasscallMetaclass is an extension of the base type.

Todo: find a good name for this metaclass.

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

```python
sage: from sage.misc.classcall_metaclass import ClasscallMetaclass
sage: class Foo(metaclass=ClasscallMetaclass):
    pass
sage: x = Foo(); x
<__main__.Foo object at 0x...>
sage: issubclass(Foo, object)
True
```

(continues on next page)


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
class Foo(object): pass
typecall(Foo)
<__main__.Foo object at 0x...>
typecall(list)
[]
typecall(Integer, 2)
2
```

Warning: typecall() doesn’t work for old style class (not inheriting from object):

```python
class Bar: pass
typecall(Bar)  # py2
Traceback (most recent call last):
...TypeError: Argument 'cls' has incorrect type (expected type, got classobj)
```

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

We illustrate some timing when using the classcall mechanism.

EXAMPLES:

```python
from sage.misc.classcall_metaclass import (ClasscallMetaclas,
ClasscallMetaclas, CRef, C2, C3, C2C, timeCall)
timeCall(object, 1000)
```

For reference let construct basic objects and a basic Python class:

```python
i1 = int(1); i3 = int(3)
# don't use Sage's Integer
class PRef(object):
    def __init__(self, i):
        self.i = i+i1
```

For a Python class, compared to the reference class there is a 10% overhead in using ClasscallMetaclas if there is no classcall defined:
sage: class P(metaclass=ClasscallMetaclass):
....:     def __init__(self, i):
....:         self.i = i+i1

sage: %timeit timeCall(PRef, 1000, i3)  # not tested
625 loops, best of 3: 420 µs per loop
sage: %timeit timeCall(P, 1000, i3)      # not tested
625 loops, best of 3: 458 µs per loop

For a Cython class (not cdef since they doesn’t allows metaclasses), the overhead is a little larger:

sage: %timeit timeCall(CRef, 1000, i3)  # not tested
625 loops, best of 3: 266 µs per loop
sage: %timeit timeCall(C2, 1000, i3)     # not tested
625 loops, best of 3: 298 µs per loop

Let’s now compare when there is a classcall defined:

sage: class PC(object, metaclass=ClasscallMetaclass):
....:     @staticmethod
....:     def __classcall__(cls, i):
....:         return i+i1

sage: %timeit timeCall(C2C, 1000, i3)  # not tested
625 loops, best of 3: 148 µs per loop
sage: %timeit timeCall(PC, 1000, i3)   # not tested
625 loops, best of 3: 289 µs per loop

The overhead of the indirection ( C(...) -> ClasscallMetaclass.__call__(...) -> C.__classcall__(...)) is unfortunately quite large in this case (two method calls instead of one). In reasonable usecases, the overhead should be mostly hidden by the computations inside the classcall:

sage: %timeit timeCall(C2C.__classcall__, 1000, C2C, i3)  # not tested
625 loops, best of 3: 33 µs per loop
sage: %timeit timeCall(PC.__classcall__, 1000, PC, i3)    # not tested
625 loops, best of 3: 131 µs per loop

Finally, there is no significant difference between Cython’s V2 and V3 syntax for metaclass:

sage: %timeit timeCall(C2, 1000, i3)  # not tested
625 loops, best of 3: 330 µs per loop
sage: %timeit timeCall(C3, 1000, i3)  # not tested
625 loops, best of 3: 328 µs per loop

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 trac ticket #18329

**class** `sage.misc.inherit_comparison.InheritComparisonClasscallMetaclas`ss

Bases: `sage.misc.classcall_metaclass.ClasscallMetaclas`, `sage.misc.inherit_comparison.InheritComparisonMetaclas`

Combine `ClasscallMetaclas` with `InheritComparisonMetaclas`.

**class** `sage.misc.inherit_comparison.InheritComparisonMetaclas`ss

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: cython(''
....: from sage.misc.inherit_comparison cimport InheritComparisonMetaclas
....: cdef class Base(object):
....:     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):
....:     def __getmetaclass__(_):
....:         from sage.misc.inherit_comparison import InheritComparisonMetaclas
....:         return InheritComparisonMetaclas
....:     def __hash__(self):
....:         return 1
....: '')
sage: a = Derived()
sage: a == a
True
sage: b = DerivedWithRichcmp()
sage: b == b
Calling Base.__richcmp__
True
```

64 Chapter 2. Programming Utilities
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
class sage.misc.method_decorator.MethodDecorator(f):
    Bases: sage.structure.sage_object.SageObject

EXAMPLES:

sage: from sage.misc.method_decorator import MethodDecorator
sage: class Foo:
    ....:     @MethodDecorator
    ....:     def bar(self, x):
    ....:         return x**2
sage: J = Foo()
sage: J.bar
<sage.misc.method_decorator.MethodDecorator object at ...>
```

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: sage.misc.fast_methods.WithEqualityById

A base class for singletons.

A singleton is a class that allows to create not more than a single instance. This instance can also belong to a subclass, but it is not possible to have several subclasses of a singleton all having distinct unique instances.

In order to create a singleton, just add Singleton to the list of base classes:

```
sage: from sage.misc.fast_methods import Singleton
sage: class C(Singleton, SageObject):
    ....:     def __init__(self):
    ....:         print("creating singleton")
sage: c = C()
creating singleton
sage: c2 = C()
sage: c is c2
True
```

The unique instance of a singleton stays in memory as long as the singleton itself does.

Pickling, copying, hashing, and comparison are provided for by Singleton according to the singleton paradigm.

Note that pickling fails if the class is replaced by a sub-sub-class after creation of the instance:

```
sage: class D(C):
    ....:     pass
sage: import __main__  # This is only needed ...
sage: __main__.C = C  # ... in doctests
sage: __main__.D = D  # same here, only in doctests
sage: orig = type(c)
sage: c.__class__ = D
sage: orig == type(c)
False
sage: loads(dumps(c))
Traceback (most recent call last):
...
AssertionError: <class '__main__.D'> is not a direct subclass of <class 'sage.misc.fast_methods.Singleton'>
```

class sage.misc.fast_methods.WithEqualityById
Bases: object

Provide hash and equality test based on identity.

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

EXAMPLES:
Any instance of `UniqueRepresentation` inherits from `WithEqualityById`.

```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 `WithEqualityById` provides unique representation behaviour:

```python
sage: a = MyUniqueParent(1)
sage: b = MyUniqueParent(2)
sage: c = MyUniqueParent(1)
sage: a is c
True
sage: d = MyUniqueParent(-1)
sage: a == d
False
```

The hash inherited from `MyParent` is replaced by a hash that coincides with `object`’s hash:

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

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

```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
```
2.1.10 Attribute and method calling

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

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

```
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
```
```

2.2 Lists and Iteration, etc.

2.2.1 Callable dictionaries

```
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.__getitem__ 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.__getitem__ (see trac ticket #6484 for benchmarks, and trac ticket #18330 for potential for improvement).

EXAMPLES:
```
In case the input is not in the dictionary, a `ValueError` is raised, for consistency with the function call syntax:

```python
sage: d[1]
Traceback (most recent call last):
  ...  
KeyError: 1
sage: 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 model 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
sage: from sage.misc.converting_dict import KeyConvertingDict
sage: d = KeyConvertingDict(int)
sage: d['3'] = 42
sage: 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: K.<x,y> = QQ[]
sage: I = ideal([x^2+y-5,x+y+3])
sage: V = sorted(I.variety(AA), key=str)
sage: v = V[0]
sage: v['x'], v['y']
(-2.464101615137755?, -0.535898384862246?)
sage: list(v)[0].parent()
Multivariate Polynomial Ring in x, y over Algebraic Real Field
```

```python
class sage.misc.converting_dict.KeyConvertingDict(key_conversion_function, data=None)
   Bases: dict

   A dictionary which automatically applies a conversions 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:

```python
sage: from sage.misc.converting_dict import KeyConvertingDict
sage: d = KeyConvertingDict(int)
sage: d['3'] = 42
sage: list(d.items())
[(3, 42)]
sage: d[5.0] = 64
sage: d['05']
64
```

**has_key**(key)

Deprecated; present just for the sake of compatibility.

Use `key in self` instead.

INPUT:

- **key** – A value identifying the element, will be converted.

EXAMPLES:

```python
sage: from sage.misc.converting_dict import KeyConvertingDict
sage: d = KeyConvertingDict(int)
sage: d[3] = 42
sage: d.has_key('3')
True
sage: d.has_key(4)
False
```

**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
sage: 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
```
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:

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

sage.misc.flatten.flatten(in_list, ltypes=(<class 'list'>, <class 'tuple'>), max_level=9223372036854775807)

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:

sage: flatten([[1,1],[1],2])
[1, 1, 1, 2]
sage: flatten([[1,2,3], (4,5), [[[1],[2]]]],max_level=1)
[1, 2, 3, 4, 5, [1], [2]]
sage: flatten([[1,2,3], (4,5), [[[1],[2]]]],max_level=0)
[[[3],[[]]]]
sage: flatten([[1,2,3], (4,5), [[[1],[2]]]],max_level=2)
[[3],[[]]]

In the following example, the vector is not flattened because it is not given in the ltypes input.

sage: flatten(('Hi',2,vector(QQ,[1,2,3]),(4,5,6)))
['Hi', 2, (1, 2, 3), 4, 5, 6]

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

sage: tV = sage.modules.vector_rational_dense.Vector_rational_dense
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.

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:
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.

```python
sage.misc.search.search(v, x)
```

Return (True, i) where i is such that v[i] == x if there is such an i, or (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:**

```python
sage: from sage.misc.search import search
sage: search([1, 4, 6, 7, 8], 6)
(True, 2)
sage: search([1, 4, 6, 7, 8], 5)
(False, 2)
sage: 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

```python
sage.misc.mrange.cantor_product(*args, **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:
sage: from itertools import islice
sage: list(islice(cantor_product(ZZ, QQ), 14r))
[(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)]

sage.misc.mrange.cartesian_product_iterator(X)
Iterate over the Cartesian product.

INPUT:

• X - list or tuple of lists

OUTPUT: iterator over the Cartesian product of the elements of X

EXAMPLES:
sage: list(cartesian_product_iterator([1,2], ['a','b']))
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]sage: list(cartesian_product_iterator([]))
[]

sage.misc.mrange.mrange(sizes, typ=<class 'list'>)
Return the multirange list with given sizes and type.
This is the list version of xmrang. Use xmrang for the iterator.

More precisely, return the iterator over all objects of type typ of n-tuples of Python ints with entries between 0 and the integers in the sizes list. The iterator is empty if sizes is empty or contains any non-positive integer.

INPUT:
• sizes - a list of nonnegative integers
• typ - (default: list) a type or class; more generally, something that can be called with a list as input.

OUTPUT: a list

EXAMPLES:

```
sage: mrange([3,2])
[[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]
sage: mrange([3,2], tuple)
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
sage: mrange([3,2], sum)
[0, 1, 1, 2, 2, 3]
```

Examples that illustrate empty multi-ranges:

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

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

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

AUTHORS:
• Jon Hanke
• William Stein

`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:

```
sage: mrange_iter([range(3),[0,2]])
[[0, 0], [0, 2], [1, 0], [1, 2], [2, 0], [2, 2]]
sage: mrange_iter([['Monty','Flying'],['Python','Circus']], tuple)
[('Monty', 'Python'), ('Monty', 'Circus'), ('Flying', 'Python'), ('Flying', 'Circus')]
sage: mrange_iter([[2,3,5,7],[1,2]], sum)
[3, 4, 4, 5, 6, 7, 8, 9]
```

Examples that illustrate empty multi-ranges:
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.

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate

generate
Next we compute the product:

```python
sage: list(xmrange([3,2], prod))
[0, 0, 0, 1, 0, 2]
```

Examples that illustrate empty multi-ranges.

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

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

```python
sage: 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')
```

AUTHORS:

• Jon Hanke
• William Stein

```python
class sage.misc.mrange.xmrange_iter(iter_list, typ=list)
Bases: object
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.
We illustrate a few more iterations.

```python
sage: list(xmrange_iter([range(3),range(2)]))
[[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]
sage: list(xmrange_iter([range(3),range(2)], tuple))
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
```

Here we compute the sum of each element of the multi-range iterator:

```python
sage: list(xmrange_iter([range(3),range(2)], sum))
[0, 1, 1, 2, 2, 3]
```

Next we compute the product:

```python
sage: list(xmrange_iter([range(3),range(2)], prod))
[0, 0, 0, 1, 0, 2]
```

Examples that illustrate empty multi-ranges.

```python
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 trac ticket #6561.

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

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_iter([X, Y], tuple):
    ....:     print((i, j))
('red', 'orange')
('red', 'horse')
('apple', 'orange')
('apple', 'horse')
(389, 'orange')
(389, 'horse')
```

AUTHORS:
• Joel B. Mohler

cardinality()

Return the cardinality of this iterator.

EXAMPLES:

```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
```

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: txt = "This monkey really likes the bananas."
sage: dic = {'monkey': 'penguin', 'bananas': 'fish'}
sage: 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:

```python
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:

```python
sage: map_threaded(lambda x: x^2, matrix([[1,2], [3,4]]))
[ 1  4]
[ 9 16]
```
AUTHORS:


### 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.

```
sage.arith.srange.ellipsis_iter(step=None, *args)
```

Same as `ellipsis_range`, but as an iterator (and may end with an Ellipsis).

See also `ellipsis_range`.

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

**EXAMPLES:**

```
sage: A = ellipsis_iter(1,2,Ellipsis)
sage: [next(A) for _ in range(10)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: next(A)
11
sage: A = ellipsis_iter(1,3,5,Ellipsis)
sage: [next(A) for _ in range(10)]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
sage: A = ellipsis_iter(1,2,Ellipsis,5,10,Ellipsis)
sage: [next(A) for _ in range(10)]
[1, 2, 3, 4, 5, 10, 11, 12, 13, 14]
```

```
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:**

```
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)
[]
```

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

Return a list of numbers `start`, `start+step`, \ldots, `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 \texttt{srange} to distinguish it from the built-in Python \texttt{range} command. The s at the beginning of the name stands for “Sage”.

See also:
\texttt{xsrage()} – iterator which is used to implement \texttt{srange()}.  

EXAMPLES:

\begin{verbatim}sage: v = srange(5); v
[0, 1, 2, 3, 4]
sage: type(v[2])
<type 'sage.rings.integer.Integer'>
sage: srange(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: srange(10, 1, -1)
[10, 9, 8, 7, 6, 5, 4, 3, 2]
sage: srange(10,1,-1, include_endpoint=True)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
sage: srange(1, 10, universe=RDF)
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
sage: srange(1, 10, 1/2)
[1, 3/2, 2, 5/2, 3, 7/2, 4, 9/2, 5, 11/2, 6, 13/2, 7, 15/2, 8, 17/2, 9, 19/2]
sage: srange(0, 1, 0.4)
[0.0, 0.4]
sage: srange(1.0, 5.0, include_endpoint=True)
[1.0, 2.0, 3.0, 4.0, 4.500000000000000]
sage: srange(1.0, 1.1)
[1.0, 1.1]
sage: srange(1.0, 1.0)
[]
sage: V = VectorSpace(QQ, 2)
\end{verbatim}
 Including the endpoint:

```
sage: srange(0, 10, step=2, include_endpoint=True)
[0, 2, 4, 6, 8, 10]
sage: srange(0, 10, step=3, include_endpoint=True)
[0, 3, 6, 9]
```

Try some inexact rings:

```
sage: srange(0.5, 1.1, 0.1, universe=RDF, include_endpoint=False)
[0.5, 0.6, 0.7, 0.7999999999999999, 0.8999999999999999, 0.9999999999999999]
sage: srange(0.5, 1, 0.1, universe=RDF, include_endpoint=False)
[0.5, 0.6, 0.7, 0.7999999999999999, 0.8999999999999999]
sage: srange(0.5, 0.9, 0.1, universe=RDF, include_endpoint=False)
[0.5, 0.6, 0.7, 0.7999999999999999]
sage: srange(0, 1.1, 0.1, universe=RDF, include_endpoint=True)
[0.0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6, 0.7, 0.7999999999999999, 0.8999999999999999, 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]
```

More examples:

```
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)
[1.0000000000000000, 1.5000000000000000, 2.0000000000000000, 2.5000000000000000, 3.0000000000000000, 3.5000000000000000, 4.0000000000000000, 4.5000000000000000]
sage: srange(0, 1, 0.4)
[0.0000000000000000, 0.4000000000000000, 0.8000000000000000]
```

Negative steps are also allowed:

```
sage: srange(4, 1, -1)
[4, 3, 2]
sage: srange(4, 1, -2/3)
[4, 7/3, 2, 5/3, 2/3]
```

```
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)
• 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: \text{1e-5})

OUTPUT: iterator

Unlike \texttt{range()}, start and end can be any type of numbers, and the resulting iterator involves numbers of that type.

\textbf{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:

\begin{verbatim}
\texttt{sage: sum([1.1] * 10) == 1.1 * 10}
False
\end{verbatim}

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 \text{endpoint_tolerance} of being integral, it is considered an exact hit, thus avoiding spurious values falling just below the endpoint.

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: xsrange(10)}
<generator object at 0x...>
\texttt{sage: for i in xsrange(1,5):
...: print(i)}
1
2
3
4
\end{verbatim}

See \texttt{srange()} for more examples.

\subsection*{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.

\begin{verbatim}
class sage.misc.element_with_label.ElementWithLabel(element, label)
    Bases: object

    Auxiliary class for showing/viewing \texttt{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.
\end{verbatim}
EXAMPLES:

```
sage: P = Poset({1: [2, 3]})
sage: labs = {i: P.rank(i) for i in range(1, 4)}
sage: print(labs)
{1: 0, 2: 1, 3: 1}
sage: print(P.plot(element_labels=labs))
Graphics object consisting of 6 graphics primitives
```

```
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)]
```

## 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 trac ticket #13579.
- Jeroen Demeyer (2013-03-17): add `atomic_write`, see trac ticket #14292.

```python
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 overwriten.
- `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:

```
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()

sage: with open(target_file, 'r') as f2:
       f2.read()
'Old contents'

sage: 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()
sage: with open(target_file, 'w') as f:
       _ = f.write("Old contents")

sage: with atomic_write(target_file) as f:
       f.close()

sage: with open(f.name, 'w') as f2:
       _ = f2.write("Newer contents")

sage: 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):
...
RuntimeError

sage: with open(target_file, 'r') as f:
       f.read()
'Newer contents'
```

Some examples of using the `append` option. Note that the file is never opened in “append” mode, it is possible to overwrite existing data:

```
sage: target_file = tmp_filename()

sage: with atomic_write(target_file, append=True) as f:
       _ = f.write("Hello")

sage: with atomic_write(target_file, append=True) as f:
       _ = f.write(" World")

sage: with open(target_file, 'r') as f:
       f.read()
'Hello World'
```

(continues on next page)
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:

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'

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'

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'

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:

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:

(continues on next page)
Supplying an encoding in binary mode (or other arguments that don’t make sense to \texttt{io.open} in binary mode) is an error:

```
sage: writer = atomic_write(target_file, binary=True,
......: encoding='utf-8')
```

```
sage: with writer as f:
......: _ = f.write('Hello')
```

```
Traceback (most recent call last):
...
ValueError: binary mode doesn't take an encoding argument
```

```
sage: os.path.exists(writer.tempname)
False
```

```
sage.misc.temporary_file.delete_tmpfiles()
```
Remove the directory SAGE_TMP.

```
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')
```

Temporary directories are unaccessible by other users:

```
sage: os.stat(d).st_mode & 0o077
0
sage: f.close()
```

```
sage.misc.temporary_file.tmp_filename(name='mp_', ext='')
```
Create and return a temporary file in 
\$HOME/.sage/temp/hostname/pid/
The temporary file is deleted automatically when Sage exits.
**Warning:** If you need a particular file extension always use `tmp_filename(ext=".foo")`, this will ensure that the file does not yet exist. If you were to use `tmp_filename()+".foo"`, then you might overwrite an existing file!

INPUT:
- `name` – (default: "tmp ") A prefix for the file name.
- `ext` – (default: "") A suffix for the file name. If you want a filename extension in the usual sense, this should start with a dot.

OUTPUT:
The absolute path of the temporary file created.

EXAMPLES:
```
sage: fn = tmp_filename('just_for_testing_', '.extension')
sage: fn  # random
'/home/username/.sage/temp/hostname/8044/just_for_testing_tVVHsn.extension'
sage: f = open(fn, 'w')
```

Temporary files are unaccessible by other users:
```
sage: os.stat(fn).st_mode & 0o077
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:
creates a file in the temp directory and returns the absolute path to that file.

EXAMPLES:
```
sage: url = 'http://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)
```
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

`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
sage: sage.misc.messaging.pushover_defaults["user"] = "USER_TOKEN"
sage: sage.misc.messaging.pushover("Hi, how are you?")  # not tested
```

Note: You may want to populate `sage.misc.messaging.pushover_defaults` with default values such as the default user in `~/.sage/init.sage`. 

2.3. File and OS Access 89
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: from sage.env import SAGE_VENV
sage: have_program('sage', os.path.join(SAGE_VENV, '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', os.path.join(SAGE_VENV, 'bin'))
False
```

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:

```python
sage: from sage.misc.sage_ostools import redirection
sage: fn = tmp_filename()

sage: with redirection(sys.stdout, open(fn, 'w')):
    ....:     print("hello world!")
```

```python
sage: with open(fn) as f:
    ....:     _ = sys.stdout.write(f.read())
hello world!
```
We can do the same using a file descriptor as source:

```python
sage: fd = sys.stdout.fileno()
```

```python
sage: with redirection(fd, open(fn, 'wb')):
    ....:    _ = os.write(fd, b"hello world!\n")
```

```python
sage: with open(fn) as f:
    ....:    _ = sys.stdout.write(f.read())
```

```
hello world!
```

The converse also works:

```python
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
```

The same `redirection` instance can be reused multiple times, provided that `close=False`:

```python
sage: f = open(fn, 'w+)
```

```python
sage: r = redirection(sys.stdout, f, close=False)
```

```python
sage: with r:
    ....: print("Line 1")
```

```python
sage: with r:
    ....: print("Line 2")
```

```python
sage: with f:
    ....: _ = f.seek(0)
    ....: _ = sys.stdout.write(f.read())
```

```
Line 1
Line 2
```

The redirection also works for subprocesses:

```python
sage: import subprocess
```

```python
sage: with redirection(sys.stdout, open(fn, 'w')):
    ....:    _ = subprocess.call(["echo", "hello world"])
```

```python
sage: with open(fn) as f:
    ....:    _ = sys.stdout.write(f.read())
```

```
hello world
```

```
dest_fd
dest_file
dup_source_fd
source_fd
source_file
```

`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: import os
sage: from sage.misc.sage_ostools import restore_cwd
sage: from sage.misc.misc import SAGE_TMP
sage: cwd = os.getcwd()

sage: with restore_cwd(str(SAGE_TMP)):
....:     print(os.getcwd() == os.path.realpath(SAGE_TMP))
True
sage: cwd == os.getcwd()
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<br>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': 'tblname', '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

class sage.database.sql_db.SQLDatabase(filename=None, read_only=None, skeleton=None)

Bases: sage.structure.sage_object.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()
```

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
```

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}),
```

(continues on next page)
Then we create the table:

```python
sage: D.create_table('simon', table_skeleton)
sage: D.show('simon')
```

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')
```

Next, add the graph on one vertex:

```python
sage: G.add_vertex()
```

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')
```

(continues on next page)
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:

```
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
[(u'CF', u'CF'), (u'CJ', u'CJ'), (u'CL', u'CL')]
```

NOTE: The values of `display_cols` are always concatenated in intersections and unions.

Of course, we can save the database to file:

```
sage: replace_with_your_own_filepath = tmp_dir()
sage: D.save(replace_with_your_own_filepath + 'simon.db')
```

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:

```
sage: E = SQLDatabase(replace_with_your own_filepath + 'simon.db')
sage: E.show('simon')
graph6 vertices edges
------------------------------------------------------------
? 0 0
B 1 0
A? 2 0
A_ 2 1
BP? 3 0
BG 3 1
BW 3 2
Bw 3 3
C? 4 0
C^ 4 1
C_ 4 2
CF 4 3
CJ 4 3
CK 4 2
CL 4 3
```

(continues on next page)
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:

```plaintext
{‘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’)
n n_squared
--------------------------
0 0
1 0
2 0
3 0
4 0
5 0
6 0
7 0
8 0
9 0
10 0
11 0
12 0
```
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:
sage: DB = SQLDatabase()
sage: DB.create_table('simon', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
sage: DB.add_rows('simon', [(0, 0), (1, 1), (1, 2)])
sage: DB.add_rows('simon', [(0, 0), (4, 0), (5, 1)], ['b2', 'a1'])

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: DB = SQLDatabase()
sage: DB.create_table('simon', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
sage: DB.add_row('simon', (0, 1))

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_rows('simon', [(0, 0), (1, 1), (1, 2)])
sage: DB.add_rows('simon', [(0, 0), (4, 0), (5, 1)], ['b2', 'a1'])
```python
sage: cur = DB.get_cursor()
sage: (cur.execute('select * from simon')).fetchall()
[(0, 0), (1, 1), (1, 2), (0, 0), (0, 4), (1, 5)]
```

**commit()**
Commit changes to file.

**EXAMPLES:**
```python
sage: DB = SQLDatabase()
sage: DB.create_table('simon',
                {'a1': {'sql': 'bool', 'index': True, 'primary_key': False},
                'b2': {'sql': 'int', 'index': True, 'primary_key': True}})
sage: DB.add_row('simon', (0, 1))
sage: DB.add_data('simon', [(0, 0), (1, 1), (1, 2)], ['b2', 'a1'])
sage: DB.drop_column('simon', 'b2')
sage: DB.commit()
sage: DB.vacuum()
```

**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:
```python
{'col1': {'sql': 'TEXT', 'index': False, 'unique': True, 'primary_key': False}, ...
```

**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:**
```python
sage: from collections import OrderedDict
sage: D = SQLDatabase()
sage: table_skeleton = OrderedDict([
    ('graph6', {'sql': 'TEXT', 'index': True, 'primary_key': True}),
    ('vertices', {'sql': 'INTEGER'}),
    ('edges', {'sql': 'INTEGER'})])
sage: D.create_table('simon', table_skeleton)
```
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
sage: from collections import OrderedDict
sage: DB = SQLDatabase()
sage: DB.create_table('lucy', OrderedDict([  
    ....: ('id', {'sql': 'INTEGER', 'primary_key':True, 'index':True}),
    ....: ('a1', {'sql': 'bool'}),
    ....: ('b2', {'sql': 'int'})
  ]))
sage: DB.add_rows('lucy', [(0,1,1),(1,1,4),(2,0,7),(3,1,384), (4,1,978932)],['id' => 'a1','b2'])
sage: DB.show('lucy')
id  a1  b2
------------------------------------------------------------
0  1  1
1  1  4
2  0  7
3  1  384
4  1  978932
```

```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')
id  a1  b2
------------------------------------------------------------
0  1  1
1  1  4
2  0  7
```

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,))
```

(continues on next page)
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>
<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:

```sage
sage: D = SQLDatabase()
sage: D.create_table('simon', {'coll':{'sql':'INTEGER'}})
sage: D.add_row('simon', (9,))
sage: D.show('simon')
coll

<table>
<thead>
<tr>
<th>coll</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
```
sage: D.drop_data_from_table('simon')
sage: D.show('simon')
coll

```
drop_index(table_name, index_name)
Set the column index_name in table table_name to not be an index. See make_index()

EXAMPLES:

```sage
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()
```
\begin{verbatim}
{'simon': {'n': {'index': False, 'primary_key': False, 'sql': 'INTEGER', 'unique': False},
'n2': {'index': False, 'primary_key': False, 'sql': 'INTEGER', 'unique': False}}}
\end{verbatim}

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:**

\begin{verbatim}
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}}}
\end{verbatim}

drop_table(table_name)

Delete table table_name from database.

**INPUT:**

- • table_name – a string

**EXAMPLES:**

\begin{verbatim}
sage: D = SQLDatabase()
sage: D.create_table('simon', {'col1': {'sql': 'INTEGER'}})
sage: D.show('simon')
coll
---------------------------
sage: D.drop_table('simon')
sage: D.get_skeleton()
{}
\end{verbatim}

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
def get_connection(\*\*, ignore_warning=None)
    Return a pysqlite 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 alter the stored data. Set self.ignore_warnings to True in order to mute future warnings.
    sage: con = D.get_connection(True)
sage: D.ignore_warnings = True
sage: con = D.get_connection()
```

```
sage: t = con.execute('CREATE TABLE simon(n INTEGER, n2 INTEGER)')
```

```
sage: for n in range(10):
    ....:     t = con.execute('INSERT INTO simon VALUES(\%d,\%d)'\%(n,n^2))
```

```
sage: D.show('simon')
n  n2
0  0
1  1
2  4
3  9
4  16
5  25
6  36
7  49
8  64
9  81
```

```python
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:

```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_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)]
```

`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:

```python
sage: GDB = GraphDatabase()
sage: GDB.get_skeleton()  # slightly random output
{u'aut_grp': {u'aut_grp_size': { 'index': True, 'unique': False, 'primary_key': False, 'sql': u'INTEGER'}}, ...
 u'num_vertices': { 'index': True, 'unique': False, 'primary_key': False, 'sql': u'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:

```python
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}}}
```

`make_primary_key(table_name, col_name)`
Set the column col_name 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:

```python
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}}}
```

`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:
```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.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:**

```python
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', 'tag': 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=(1,))

sage: q.get_query_string()
'SELECT tag FROM simon WHERE wolf=?'

sage: 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', {'coll': {'sql': 'INTEGER'}})

sage: D.show('simon')

coll

-------------------

sage: D.rename_table('simon', 'lucy')

sage: D.show('simon')

Traceback (most recent call last):
...
RuntimeError: Failure to fetch data.

sage: D.show('lucy')

coll

---------------------
```

2.4. Database Access
**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: tmp = tmp_dir()  # replace with your own file path
sage: MonicPolys.save(tmp + 'sage.db')
sage: N = SQLDatabase(tmp + 'sage.db')
sage: N.show('simon')
```

```text
n
--------------------
0 1
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 9
9 10
10 11
11 12
12 13
13 14
14 15
15 16
16 17
17 18
18 19
```

**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')
```

```text
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
class sage.databases.sql_db.SQLQuery(database, *args, **kwds)
Bases: sage.structure.sage_object.SageObject

A query for a SQLite database.

INPUT:

- database – a SQLDatabase object
- query_dict – a dictionary specifying the query itself. The format is:

  ```python
  {'table_name': 'tblname', 'display_cols': ['col1', 'col2', 'col3'], 'expression': \[
  \text{\(col\), 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: r = SQLQuery(D, {'table_name': 'simon', 'display_cols': ['a1'], 'expression': ['b2', '<=', 3]})
sage: r.show()
a1
0
1
```

Test that trac ticket #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
```

```python
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):  
(continues on next page)
```
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

get_query_string()

Return a copy of the query string.

EXAMPLES:

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'
sage: 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 if the two queries only search on 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).
- 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]})
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)
sage: Q.query_results()
[(18, u'D??'), (19, u'D?C'), (20, u'D?K'), (21, u'D@O'), (22, u'D['),
 (23, u'C?'), (24, u'C@'), (25, u'CB'), (26, u'CK'), (27, u'CF'), (28, u'CJ'),
 (29, u'CL'), (30, u'CN'), (31, u'C'), (32, u'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:
• **id_col** – reference to a column that can be used as an object identifier for each row

**EXAMPLES:**

```python
def SQLDatabase():
    DB = SQLDatabase()
sage: DB = SQLDatabase()
sage: DB.create_table('simon',
                   [{'sql': 'bool', 'primary_key': False},
                    {'sql': 'int'}])

sage: DB.add_data('simon', [(0, 0), (1, 1), (1, 2)])
sage: r = SQLQuery(DB, {'table_name': 'simon',
                      'display_cols': ['a1'],
                      'expression': ['b2', '<=', 6]})

sage: r.show()
a1
--------------------
  0
  1
  1
```

```python
def GraphDatabase():
    D = GraphDatabase()
sage: from sage.graphs.graph_database import valid_kwds, data_to_degseq
sage: relabel = {}
sage: for col in valid_kwds:
    ...:     relabel[col] = 
    ...:         .join([word.capitalize() for word in col.split('_')])

sage: q = GraphQuery(display_cols=['graph6', 'degree_sequence'],
                   num_vertices=4)

sage: 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>Degree Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph6</td>
</tr>
<tr>
<td>[0, 0, 0, 0]</td>
</tr>
<tr>
<td>[0, 0, 1, 1]</td>
</tr>
<tr>
<td>[0, 1, 1, 2]</td>
</tr>
<tr>
<td>[1, 1, 1, 3]</td>
</tr>
<tr>
<td>[0, 2, 2, 2]</td>
</tr>
<tr>
<td>[1, 1, 2, 2]</td>
</tr>
<tr>
<td>[1, 2, 2, 3]</td>
</tr>
<tr>
<td>[2, 2, 2, 2]</td>
</tr>
<tr>
<td>[2, 2, 3, 3]</td>
</tr>
<tr>
<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 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 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:
```python
{'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.union(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 = ? ) OR ( simon.b2 <= ? )' 
sage: s.query_results()
[(1, 1), (4, 1)]
```

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:
```
sage: G = SQLDatabase(GraphDatabase().__dblocation__, False)
sage: from sage.databases.sql_db import construct_skeleton
sage: sorted(construct_skeleton(G))
[u'aut_grp', u'degrees', u'graph_data', u'misc', u'spectrum']
```

sage.databases.sql_db.regexp(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:

```
sage: from sage.databases.sql_db import regexp
sage: regexp('.*','cs')
True
sage: regexp('.*','ccs')
False
sage: regexp('.*','cccc')
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:

```
{ 'primary_key':False, 'index':False, 'unique': False, 'sql':'REAL'}
```

EXAMPLES:

```
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': True, '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'}. 
```

`sage.databases.sql_db.verify_operator(operator)`
Check that `operator` is one of the allowed strings. Legal operators include the following strings:

• ‘=’
• ‘<=’
• ‘>=’
• ‘<’
• ‘>’
• ‘<>’
• ‘like’
• ‘regexp’
• ‘is null’
• ‘is not null’

EXAMPLES:
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.

sage.databases.sql_db.verify_type(type)
Verify that the specified type is one of the allowed strings.

EXAMPLES:

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.

2.5 Media

2.5.1 Wrapper for Graphics Files

class sage.structure.graphics_file.GraphicsFile(filename, mime_type=None)
   Bases: sage.structure.sage_object.SageObject
   Wrapper around a graphics file.

data()
   Return a byte string containing the image file.

filename()

launch_viewer()
   Launch external viewer for the graphics file.

   Note: Does not actually launch a new process when doctesting.

   EXAMPLES:

   sage: from sage.structure.graphics_file import GraphicsFile
   sage: g = GraphicsFile('/tmp/test.png', 'image/png')
   sage: g.launch_viewer()

mime()
sagenb_embedding()
Embed in SageNB
This amounts to just placing the file in the cell directory. The notebook will then try to guess what we want with it.

save_as(filename)
Make the file available under a new filename.

INPUT:
• filename – string. The new filename.

The newly-created filename will be a hardlink if possible. If not, an independent copy is created.

class sage.structure.graphics_file.Mime
Bases: object

classmethod extension(mime_type)
Return file extension.

INPUT:
• mime_type – mime type as string.

OUTPUT:
String containing the usual file extension for that type of file. Excludes os.extsep.

EXAMPLES:

```
 sage: from sage.structure.graphics_file import Mime
 sage: Mime.extension('image/png')
 'png'
```

classmethod validate(value)
Check that input is known mime type

INPUT:
• value – string.

OUTPUT:
Unicode string of that mime type. A ValueError is raised if input is incorrect / unknown.

EXAMPLES:

```
 sage: from sage.structure.graphics_file import Mime
 sage: Mime.validate('image/png')
 u'image/png'
 sage: Mime.validate('foo/bar')
 Traceback (most recent call last):
 ... ValueError: unknown mime type
```

sage.structure.graphics_file.graphics_from_save(save_function, preferred_mime_types, allowed_mime_types=None, figsize=None, dpi=None)

Helper function to construct a graphics file.

INPUT:
• **save_function** – callable that can save graphics to a file and accepts options like `sage.plot.graphics.Graphics.save()`.

• **preferred_mime_types** – list of mime types. The graphics output mime types in order of preference (i.e. best quality to worst).

• **allowed_mime_types** – set of mime types (as strings). The graphics types that we can display. Output, if any, will be one of those.

• **figsize** – pair of integers (optional). The desired graphics size in pixels. Suggested, but need not be respected by the output.

• **dpi** – integer (optional). The desired resolution in dots per inch. Suggested, but need not be respected by the output.

**OUTPUT:**

Return an instance of `sage.structure.graphics_file.GraphicsFile` encapsulating a suitable image file. Image is one of the `preferred_mime_types`. If `allowed_mime_types` is specified, the resulting file format matches one of these.

Alternatively, this function can return `None` to indicate that textual representation is preferable and/or no graphics with the desired mime type can be generated.

### 2.5.2 Work with WAV files

A WAV file is a header specifying format information, followed by a sequence of bytes, representing the state of some audio signal over a length of time.

A WAV file may have any number of channels. Typically, they have 1 (mono) or 2 (for stereo). The data of a WAV file is given as a sequence of frames. A frame consists of samples. There is one sample per channel, per frame. Every wav file has a sample width, or, the number of bytes per sample. Typically this is either 1 or 2 bytes.

The wav module supplies more convenient access to this data. In particular, see the docstring for `Wave.channel_data()`.

The header contains information necessary for playing the WAV file, including the number of frames per second, the number of bytes per sample, and the number of channels in the file.

**AUTHORS:**

- William Stein (2007-07-03): add more
- Bobby Moretti (2007-07-03): add doctests

**class** `sage.media.wav.Wave(data=None, **kwds)`

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

A class wrapping a wave audio file.

**INPUT:**

You must call `Wave()` with either `data = filename`, where `filename` is the name of a wave file, or with each of the following options:

- **channels** – the number of channels in the wave file (1 for mono, 2 for stereo, etc…)
- **width** – the number of bytes per sample
- **framerate** – the number of frames per second
- **nframes** – the number of frames in the data stream
• bytes – a string object containing the bytes of the data stream

Slicing:

Slicing a Wave object returns a new wave object that has been trimmed to the bytes that you have
given it.

Indexing:

Getting the $n$th item in a Wave object will give you the value of the $n$th frame.

channel_data($n$)
Get the data from a given channel.

INPUT:

$n$ – the channel number to get

OUTPUT:

A list of signed ints, each containing the value of a frame.

convolve($right$, $channel=0$)

CONVOLVE

NOT DONE!

Convolution of self and other, i.e., add their fft’s, then inverse fft back.

domain($npoints=None$)

Used internally for plotting. Get the x-values for the various points to plot.

getframerate()

Return the number of frames per second in this wave object.

OUTPUT:

The frame rate of this sound file.

getlength()

Return the length of this file (in seconds).

OUTPUT:

The running time of the entire WAV object.

getnchannels()

Return the number of channels in this wave object.

OUTPUT:

The number of channels in this wave file.

getnframes()

Return the total number of frames in this wave object.

OUTPUT:

The number of frames in this WAV.

getsampwidth()

Return the number of bytes per sample in this wave object.

OUTPUT:

The number of bytes in each sample.

listen()

Listen to (or download) this wave file.
Creates a link to this wave file in the notebook.

```
plot(npoints=None, channel=0, plotjoined=True, **kwds)
```

Plots the audio data.

**INPUT:**

- `npoints` – number of sample points to take; if not given, draws all known points.
- `channel` – 0 or 1 (if stereo). default: 0
- `plotjoined` – whether to just draw dots or draw lines between sample points

**OUTPUT:**

a plot object that can be shown.

```
plot_fft(npoints=None, channel=0, half=True, **kwds)
plot_raw(npoints=None, channel=0, plotjoined=True, **kwds)
```

**readframes(n)**

Read out the raw data for the first `n` frames of this wave object.

**INPUT:**

- `n` – the number of frames to return

**OUTPUT:**

A list of bytes (in string form) representing the raw wav data.

```
save(filename='sage.wav')
```

Save this wave file to disk, either as a Sage sobj or as a .wav file.

**INPUT:**

- `filename` – the path of the file to save. If filename ends with `wav`, then save as a wave file, otherwise, save a Sage object.

If no input is given, save the file as ‘sage.wav’.

```
set_values(values, channel=0)
```

Used internally for plotting. Get the y-values for the various points to plot.

```
slice_seconds(start, stop)
```

Slice the wave from start to stop.

**INPUT:**

- `start` – the time index from which to begin the slice (in seconds)
- `stop` – the time index from which to end the slice (in seconds)

**OUTPUT:**

A Wave object whose data is this object’s data, sliced between the given time indices.

```
values(npoints=None, channel=0)
```

Used internally for plotting. Get the y-values for the various points to plot.

```
vector(npoints=None, channel=0)
```

2.5. Media
2.6 Warnings

2.6.1 Stopgaps

exception sage.misc.stopgap.StopgapWarning

Bases: Warning

This class is used to warn users of a known issue that may produce mathematically incorrect results.

sage.misc.stopgap.set_state(mode)

Enable or disable the stopgap warnings.

INPUT:

- mode – (bool); if True, enable stopgaps; otherwise, disable.

EXAMPLES:

```
sage: import sage.misc.stopgap
sage: sage.misc.stopgap.set_state(False)
sage: sage.misc.stopgap.stopgap("Displays nothing.", 12345)
```

```
sage: sage.misc.stopgap.set_state(True)
```

```
sage: sage.misc.stopgap.stopgap("Displays something.", 123456)
doctest:...:
...StopgapWarning: Displays something.
This issue is being tracked at https://trac.sagemath.org/sage_trac/ticket/123456.
sage: sage.misc.stopgap.set_state(False)
```

sage.misc.stopgap.stopgap(message, ticket_no)

Issue a stopgap warning.

INPUT:

- message - an explanation of how an incorrect answer might be produced.
- ticket_no - an integer, giving the number of the Trac ticket tracking the underlying issue.

EXAMPLES:

```
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://trac.sagemath.org/sage_trac/ticket/12509.
sage: sage.misc.stopgap.stopgap("This is not printed", 12509)
sage: sage.misc.stopgap.set_state(False)  # so rest of doctesting fine
```
2.6.2 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 trac ticket number trac ticket #13109, which is where this mandatory argument to `deprecation()` was introduced.

Functions and classes

class sage.misc.superseded.DeprecatedFunctionAlias(trac_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(trac_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(trac_number, f)` to make a deprecated aliased version of f.

INPUT:

• `trac_number` – integer. The trac ticket number where the deprecation is introduced.
• `func` – the function or method to be aliased

EXAMPLES:

```
sage: from sage.misc.superseded import deprecated_function_alias
dsage: g = deprecated_function_alias(13109, number_of_partitions)
dsage: g(5)
doctest:...: DeprecationWarning: g is deprecated. Please use sage.combinat.
               →             partition.number_of_partitions instead.
See http://trac.sagemath.org/13109 for details.
```

This also works for methods:

```
sage: class cls(object):
    ....:     def new_meth(self): return 42
    ....:     old_meth = deprecated_function_alias(13109, new_meth)
dsage: cls().old_meth()
doctest:...: DeprecationWarning: old_meth is deprecated. Please use new_meth
               →instead.
See http://trac.sagemath.org/13109 for details.
```

trac ticket #11585:
sage: def a(): pass
sage: b = deprecated_function_alias(13109, a)
sage: b()
doctest:...: DeprecationWarning: b is deprecated. Please use a instead.
See http://trac.sagemath.org/13109 for details.

AUTHORS:

• Florent Hivert (2009-11-23), with the help of Mike Hansen.
• Luca De Feo (2011-07-11), printing the full module path when different from old path

sage.misc.superseded.deprecation(trac_number, message, stacklevel=4)
Issue a deprecation warning.

INPUT:

• trac_number – integer. The trac ticket number where the deprecation is introduced.
• message – string. An explanation why things are deprecated and by what it should be replaced.
• 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 http://trac.sagemath.org/13109 for details.

See also:

experimental(), warning().

class sage.misc.superseded.experimental(trac_number, stacklevel=4)
Bases: object
A decorator which warns about the experimental/unstable status of the decorated class/method/function.

INPUT:

• trac_number – an integer. The trac ticket number where this code was introduced.
• stack_level – (default: 4) an integer. This is passed on to warnings.warn().

EXAMPLES:

sage: @sage.misc.superseded.experimental(trac_number=79997)
....: def foo(*args, **kwargs):
....:     print("{} {}".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://trac.sagemath.org/79997 for details.
(7,) {'what': 'Hello'}
sage: class bird(SageObject):
....:     @sage.misc.superseded.experimental(trac_number=99999)
....:     def __init__(self, *args, **kwargs):
....:         print("piep {} {}".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://trac.sagemath.org/99999 for details.
piep (99,) {}

See also:

experimental(), warning(), deprecation().

sage.misc.superseded.experimental_warning(trac_number, message, stacklevel=4)
Issue a warning that the functionality or class is experimental and might change in future.

INPUT:

• trac_number – an integer. The trac ticket 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://trac.sagemath.org/66666 for details.

See also:

mark_as_experimental, warning(), deprecation().

sage.misc.superseded.warning(trac_number, message, warning_class=<class 'Warning'>, stacklevel=3)
Issue a warning.

INPUT:

• trac_number – integer. The trac ticket 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:

sage: def foo():
....:     sage.misc.superseded.warning(
....:         99999,
....:         'The syntax will change in future.','

(continues on next page)

(continued from previous page)

....:

sage: foo()

doctest:....: FutureWarning: The syntax will change in future.
See https://trac.sagemath.org/99999 for details.

See also:

deprecation(), experimental(), exceptions.Warning.

2.7 Miscellaneous Useful Functions

2.7.1 Miscellaneous functions

AUTHORS:

- William Stein
- William Stein (2006-04-26): added workaround for Windows where most users’ home directory has a space in it.

class sage.misc.misc.BackslashOperator

Bases: object

Implements Matlab-style backslash operator for solving systems:

\[ A \backslash b \]

The preparser converts this to multiplications using BackslashOperator().

EXAMPLES:

\[
\begin{align*}
\text{sage: } & \text{preparse("A } \backslash \text{ matrix(QQ,2,1,[1/3,'2/3'])")} \\
& "A * BackslashOperator() * matrix(QQ,Integer(2),Integer(1),[Integer(1)/Integer(3),
\rightarrow '2/3'])"
\end{align*}
\]

\[
\begin{align*}
\text{sage: } & \text{preparse("A } \backslash \text{ matrix(QQ,2,1,[1/3,2*3])")} \\
& 'A * BackslashOperator() * matrix(QQ,Integer(2),Integer(1),[Integer(1)/Integer(3),
\rightarrow Integer(2)*Integer(3)])'
\end{align*}
\]

\[
\begin{align*}
\text{sage: } & \text{preparse("A } \backslash \text{ B + C")} \\
& 'A * BackslashOperator() * B + C'
\end{align*}
\]

\[
\begin{align*}
\text{sage: } & \text{preparse("A } \backslash \text{ eval('C+D')")} \\
& 'A * BackslashOperator() * eval('C+D')"
\end{align*}
\]

\[
\begin{align*}
\text{sage: } & \text{preparse("A } \backslash \text{ x / 5")} \\
& 'A * BackslashOperator() * x / Integer(5)'
\end{align*}
\]

\[
\begin{align*}
\text{sage: } & \text{preparse("A^3 } \backslash \text{ b")} \\
& 'A**Integer(3) * BackslashOperator() * b'
\end{align*}
\]

class sage.misc.misc.GlobalCputime(t)

Bases: object

Container for CPU times of subprocesses.

AUTHOR:

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

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

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

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

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

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

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

See also:

cputime()

```python
sage.misc.misc.compose(f, g)
```

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

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:

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

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

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

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

The measurement for the main Sage process is done via a call to `resource.getrusage()`, so it avoids the wraparound problems in `time.clock()` on Cygwin.

**INPUT:**
- `t` - (optional) time in CPU seconds, if `t` is a result from an earlier call with `subprocesses=True`, then `subprocesses=True` is assumed.
- `subprocesses` – (optional), include subprocesses (default: False)

**OUTPUT:**
- `float` - time in CPU seconds if `subprocesses=False`
- `GlobalCputime` - object which holds CPU times of subprocesses otherwise

**EXAMPLES:**

```python
sage: t = cputime()
sage: F = gp.factor(2^199-1)
sage: cputime(t) # somewhat random 0.010999000000000092

sage: t = cputime(subprocesses=True)
sage: F = gp.factor(2^199-1)
sage: cputime(t) # somewhat random 0.091999

sage: w = walltime()
sage: F = gp.factor(2^199-1)
sage: walltime(w) # somewhat random 0.58425593376159668
```

**Note:** Even with `subprocesses=True` there is no guarantee that the CPU time is reported correctly because subprocesses can be started and terminated at any given time.

---

**sage.misc.misc.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:**

```python
sage: from sage.misc.misc import exactly_one_is_true
sage: exactly_one_is_true([]) False
```

(continues on next page)
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
sage: 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)
```

sage.misc.misc.forall($S$, $P$)

If $P(x)$ is true every $x$ in $S$, return True and None. If there is some element $x$ in $S$ such that $P$ is not True, return False and $x$.

Note that this function is NOT suitable to be used in an if-statement or in any place where a boolean expression is expected. For those situations, use the Python built-in `all(P(x) for x in S)`
INPUT:

- S - object (that supports enumeration)
- P - function that returns True or False

OUTPUT:

- bool - whether or not P is True for all elements of S
- object - x

EXAMPLES: lambda functions are very useful when using the forall function. As a toy example we test whether certain integers are greater than 3.

```
sage: forall([1,2,5], lambda x : x > 3)
(False, 1)
sage: forall([1,2,5], lambda x : x > 0)
(True, None)
```

Next we ask whether every positive integer less than 100 is a product of at most 2 prime factors:

```
sage: forall(range(1,100), lambda n : len(factor(n)) <= 2)
(False, 30)
```

The answer is no, and 30 is a counterexample. However, every positive integer 100 is a product of at most 3 primes.

```
sage: forall(range(1,100), lambda n : len(factor(n)) <= 3)
(True, None)
```

`sage.misc.misc.get_main_globals()`

Return the main global namespace.

EXAMPLES:

```
sage: from sage.misc.misc import get_main_globals
sage: G = get_main_globals()
sage: bla = 1
sage: G['bla']
1
sage: bla = 2
sage: G['bla']
2
sage: G['ble'] = 5
sage: ble
5
```

This is analogous to `globals()`, except that it can be called from any function, even if it is in a Python module:

```
sage: def f():
    ....:     G = get_main_globals()
    ....:     assert G['bli'] == 14
    ....:     G['blo'] = 42
sage: bli = 14
sage: f()
sage: blo
42
```
ALGORITHM:
The main global namespace is discovered by going up the frame stack until the frame for the __main__ module is found. Should this frame not be found (this should not occur in normal operation), an exception “ValueError: call stack is not deep enough” will be raised by _getframe.

See `inject_variable_test()` for a real test that this works within deeply nested calls in a function defined in a Python module.

```python
from sage.misc.misc import inject_variable
inject_variable("a", 314)
a 314
```

A warning is issued the first time an existing value is overwritten:

```python
inject_variable("a", 271)
doctest:...: RuntimeWarning: redefining global value `a`
inject_variable("a", 272)
a 272
```

That’s because warn seem to not reissue twice the same warning:

```python
from warnings import warn
warn("blah")
doctest:...: UserWarning: blah
```

Warnings can be disabled:

```python
b = 3
inject_variable("b", 42, warn=False)
b 42
```

Use with care!

```python
from sage.misc.misc import inject_variable_test
inject_variable_test("a0", 314, 0)
a0 314
```

(continues on next page)
sage: inject_variable_test("a1", 314, 1)
sage: a1
314
sage: inject_variable_test("a2", 314, 2)
sage: a2
314
sage: inject_variable_test("a2", 271, 2)

doctest:...: RuntimeWarning: redefining global value `a2`
sage: a2
271

sage.misc.misc.is_in_string(line, pos)

Return True if the character at position pos in line occurs within a string.

EXAMPLES:

```
sage: from sage.misc.misc import is_in_string
sage: line = 'test(#)
\n
sage: is_in_string(line, line.rfind('#'))
True
sage: is_in_string(line, line.rfind(')'))
False
```

sage.misc.misc.is_iterator(it)

Tests if it is an iterator.

The mantra if hasattr(it, 'next') was used to tests if it is an iterator. This is not quite correct since it could have a next methods with a different semantic.

EXAMPLES:

```
sage: it = iter([1,2,3])
sage: is_iterator(it)
True
sage: class wrong():
....:     def __init__(self): self.n = 5
....:     def __next__(self):
....:         self.n -= 1
....:         if self.n == 0: raise StopIteration
....:         return self.n
sage: x = wrong()
sage: is_iterator(x)
False
sage: list(x)
Traceback (most recent call last):
...
TypeError: 'wrong' object is not iterable
sage: class good(wrong):
....:     def __iter__(self): return self
sage: x = good()
```

(continues on next page)
```python
sage: list(x)
[4, 3, 2, 1]
sage: P = Partitions(3)
sage: is_iterator(P)
False
sage: is_iterator(iter(P))
True
```

```python
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
```

```python
sage.misc.misc.nest(f, n, x)
Return \( f(f(\ldots f(x)\ldots)) \), where the composition occurs \( n \) times.

See also \texttt{compose()} and \texttt{self_compose()}

INPUT:

- \( f \) – a function of one variable
- \( n \) – a nonnegative integer
- \( x \) – any input for \( f \)

OUTPUT: \( f(f(\ldots f(x)\ldots)) \), where the composition occurs \( n \) times

EXAMPLES:
```python
sage: def f(x): return x**2 + 1
sage: x = var('x')
sage: nest(f, 3, x)
((x^2 + 1)^2 + 1)^2 + 1
sage: _ = function('f')
sage: _ = var('x')
sage: nest(f, 10, x)
f(f(f(f(f(f(f(f(f(f(x))))))))))
```

```python
sage: _ = function('f')
sage: _ = var('x')
sage: nest(f, 0, x)
x
```
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:**

```
sage: newton_method_sizes(17)
[1, 2, 3, 5, 9, 17]
sage: newton_method_sizes(16)
[1, 2, 4, 8, 16]
sage: newton_method_sizes(1)
[1]
```

**AUTHORS:**
- David Harvey (2006-09-09)

sage.misc.misc.pad_zeros(s, size=3)

**EXAMPLES:**

```
sage: pad_zeros(100)
'100'
sage: pad_zeros(10)
'010'
sage: pad_zeros(10, 5)
'00010'
sage: pad_zeros(389, 5)
'00389'
sage: pad_zeros(389, 10)
'0000000389'
```

sage.misc.misc.powerset(X)
Iterator over the list of all subsets of the iterable \(X\), in no particular order. Each list appears exactly once, up to order.

**INPUT:**
- \(X\) - an iterable

**OUTPUT:** iterator of lists

**EXAMPLES:**

```
sage: list(powerset([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
sage: [z for z in powerset([0,[1,2]])]
[[], [0], [[1, 2]], [0, [1, 2]]]
```

Iterating over the power set of an infinite set is also allowed:

```
sage: i = 0
sage: L = []
```

(continues on next page)
sage: for x in powerset(ZZ):
    ....:     if i > 10:
    ....:         break
    ....:     else:
    ....:         i += 1
    ....:         L.append(x)

sage: print(" ".join(str(x) for x in L))
[] [0] [1] [0, 1] [-1] [0, -1] [1, -1] [0, 1, -1] [2] [0, 2] [1, 2]

You may also use subsets as an alias for powerset:

sage: subsets([1,2,3])
<generator object ...powerset at 0x...>
sage: list(subsets([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

The reason we return lists instead of sets is that the elements of sets must be hashable and many structures on which one wants the powerset consist of non-hashable objects.

AUTHORS:

• William Stein
• Nils Bruin (2006-12-19): rewrite to work for not-necessarily finite objects X.

sage.misc.misc.random_sublist(X, s)

Return a pseudo-random sublist of the list X where the probability of including a particular element is s.

INPUT:

• X - list
• s - floating point number between 0 and 1

OUTPUT: list

EXAMPLES:

sage: from sage.misc.misc import is_sublist
sage: S = [1,7,3,4,18]
sage: sublist = random_sublist(S, 0.5); sublist  # random
[1, 3, 4]
sage: is_sublist(sublist, S)
True
sage: sublist = random_sublist(S, 0.5); sublist  # random
[1, 3]
sage: is_sublist(sublist, S)
True

sage.misc.misc.sage_makedirs(dirname, mode=511)

Python version of mkdir -p: try to create a directory, and also create all intermediate directories as necessary. Succeed silently if the directory already exists (unlike os.makedirs()). Raise other errors (like permission errors) normally.

EXAMPLES:

```
sage: from sage.misc.misc import sage_makedirs
sage: sage_makedirs(DOT_SAGE)  # no output
```

The following fails because we are trying to create a directory in place of an ordinary file:

```
sage: filename = tmp_filename()
sage: sage_makedirs(filename)
Traceback (most recent call last):
  ...  
OSError: [Errno ...] File exists: ...
```

`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.subsets(X)`

Iterator over the list of all subsets of the iterable `X`, in no particular order. Each list appears exactly once, up to order.

**INPUT:**

- `X` - an iterable

**OUTPUT:** iterator of lists

**EXAMPLES:**

```
sage: list(powerset([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
sage: [z for z in powerset([0,[1,2]])]
[[], [0], [[1, 2]], [0, [1, 2]]]
```

Iterating over the power set of an infinite set is also allowed:

```
sage: i = 0
sage: L = []
sage: for x in powerset(ZZ):
    ....:     if i > 10:
```

(continues on next page)
...:
...: break
...:
...: else:
...:     i += 1
...: L.append(x)
sage: print(" ".join(str(x) for x in L))
[] [0] [1] [0, 1] [-1] [0, -1] [1, -1] [0, 1, -1] [2] [0, 2] [1, 2]

You may also use subsets as an alias for powerset:
sage: subsets([1,2,3])
<generator object ...powerset at 0x...>
sage: list(subsets([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

The reason we return lists instead of sets is that the elements of sets must be hashable and many structures on which one wants the powerset consist of non-hashable objects.

AUTHORS:

• William Stein

• Nils Bruin (2006-12-19): rewrite to work for not-necessarily finite objects X.

sage.misc.misc.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:

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\n'b\nc')
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
sage: with io.open(filename) as fobj:
    ....:   print(try_read(fobj))
a
b
c
```

I/O buffers:

```python
sage: buf = io.StringIO(u'a
b
c')
sage: print(try_read(buf))
a
b
c
sage: _ = buf.seek(0); try_read(buf, splitlines=True)
[u'a\n', u'b\n', u'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
', b'b
', b'c']
True
```

Custom readable:

```python
sage: class MyFile(object):
    ....:   def read(self):
    ....:       return 'Hello world!'
sage: try_read(MyFile())
'Hello world!'
sage: try_read(MyFile(), splitlines=True)
['Hello world!']
```

Not readable:

```python
sage: try_read(1) is None
True
```

```
sage.misc.misc.union(x, y=None)
```

Return the union of x and y, as a list. The resulting list need not be sorted and can change from call to call.

**INPUT:**

- x - iterable
- y - iterable (may optionally omitted)

**OUTPUT:** list

**EXAMPLES:**

```python
sage: answer = union([1,2,3,4], [5,6]); answer
[1, 2, 3, 4, 5, 6]
sage: union([1,2,3,4,5,6], [5,6]) == answer
True
```
True
sage: union((1,2,3,4,5,6), [5,6]) == answer
True
sage: union((1,2,3,4,5,6), set([5,6])) == answer
True

sage.misc.miss.uniq(x)
Return the sublist of all elements in the list x that is sorted and is such that the entries in the sublist are unique.

EXAMPLES:

sage: uniq([1, 1, 8, -5, 3, -5, -13, 13, -13])
doctest:...: DeprecationWarning: the output of uniq(X) being sorted is deprecated; →use sorted(set(X)) instead if you want sorted output
See https://trac.sagemath.org/27014 for details.
[-13, -5, 1, 3, 8, 13]

sage.misc.miss.walltime(t=0)
Return the wall time in second, or with optional argument t, return the wall time since time t. “Wall time” means the time on a wall clock, i.e., the actual time.

INPUT:
• t - (optional) float, time in CPU seconds

OUTPUT:
• float - time in seconds

EXAMPLES:

sage: w = walltime()
sage: F = factor(2^199-1)
sage: walltime(w)  # somewhat random
0.8823847770690918

sage.misc.miss.word_wrap(s, ncols=85)

2.7.2 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 (2007-10-26): Balanced product tree, other optimizations, (lazy) generator support
• Robert Bradshaw (2008-03-26): Balanced product tree for generators and iterators
• Stefan van Zwam (2013-06-06): Added bitset tests, some docstring cleanup

class sage.misc.miss_c.NonAssociative(left, right=None)
Bases: object
This class is to test the balance nature of prod.
EXAMPLES:

```python
sage: from sage.misc.misc_c import NonAssociative
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: prod(L, recursion_cutoff=1)
((((((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)))
```

```python
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:

```python
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:

```python
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:

```python
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

```python
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 trac ticket #28649.

Use this function after a doctest which produces potentially unflushed output to force it to be flushed.

EXAMPLES:
sage: R.<t> = QQ[]
sage: t^(sys.maxsize//2)
Traceback (most recent call last):
...
RuntimeError: FLINT exception
sage: from sage.misc.misc_c import cyflush
cyflush()
...

sage.misc.misc_c.iterator_prod(L, z=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:

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))

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:

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
sage: normalize_index(6,5)
Traceback (most recent call last):
  ... IndexError: index out of range
sage: normalize_index((4,-6),5)
Traceback (most recent call last):
  ... IndexError: index out of range
sage: normalize_index((5,0),5)
Traceback (most recent call last):
  ... IndexError: index out of range
sage: normalize_index((-2,3),5)
[3, 3]
sage: normalize_index((5,0),5)
Traceback (most recent call last):
  ... IndexError: index out of range
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,4); 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,-2); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(None,-2,4); 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
sage: s=slice(-2,None,4); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(-2,-2,None); normalize_index(s,5)==list(range(5))[s]
True
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(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(4,4,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
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

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:

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

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
• 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:
• L – the list
• start – (optional) a default start value

EXAMPLES:

sage: running_total(range(5))
[0, 1, 3, 6, 10]
sage: running_total("abcde")
['a', 'ab', 'abc', 'abcd', 'abcde', 'abcdef']
sage: running_total("abcdef", start="%")
['%a', '%ab', '%abc', '%abcd', '%abcde', '%abcdef']
sage: running_total([1..10], start=100)
[101, 103, 106, 110, 115, 121, 128, 136, 145, 155]
class sage.misc.misc_c.sized_iter

Bases: object

Wrapper for an iterator to verify that it has a specified length.

INPUT:

• iterable – object to be iterated over
• 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:

\begin{verbatim}
sage: from sage.misc.misc_c import sized_iter
sage: list(sized_iter(range(4)))
[0, 1, 2, 3]
\end{verbatim}

\begin{verbatim}
sage: list(sized_iter(range(4), 4))
[0, 1, 2, 3]
\end{verbatim}

\begin{verbatim}
sage: list(sized_iter(range(5), 4))
Traceback (most recent call last):
...
ValueError: sequence too long (expected length 4, got more)
\end{verbatim}

\begin{verbatim}
sage: list(sized_iter(range(3), 4))
Traceback (most recent call last):
...
ValueError: sequence too short (expected length 4, got 3)
\end{verbatim}

If the iterable is too long, we get the error on the last entry:

\begin{verbatim}
sage: it = sized_iter(range(5), 2)
sage: next(it)
0
\end{verbatim}

\begin{verbatim}
sage: next(it)
Traceback (most recent call last):
...
ValueError: sequence too long (expected length 2, got more)
\end{verbatim}

When the expected length is zero, the iterator is checked on construction:

\begin{verbatim}
sage: list(sized_iter([], 0))
[]
\end{verbatim}

\begin{verbatim}
sage: sized_iter([1], 0)
Traceback (most recent call last):
...
ValueError: sequence too long (expected length 0, got more)
\end{verbatim}

If no \texttt{length} is given, the iterable must implement \texttt{__len__}:

\begin{verbatim}
sage: sized_iter(x \texttt{for} x \texttt{in} range(4))
Traceback (most recent call last):
\end{verbatim}
2.7.3 Verbosity System and Logging in SageMath

Howto: Logging

Using Python’s Logging Module

Import it:

```python
sage: import logging
def logging.basicConfig()  # only needed once
```

Setting the level:

```python
def logging.getLogger().setLevel(logging.INFO)
```

Log something:

```python
do logger = logging.getLogger(__name__)
do 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
do 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
do logging.getLogger().setLevel(logging.WARNING)
```

Warnings are still shown at this default level (`logging.WARNING`):

```python
do 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>logging.CRITICAL</td>
</tr>
<tr>
<td>−1</td>
<td>logging.ERROR</td>
</tr>
<tr>
<td>0</td>
<td>logging.WARNING</td>
</tr>
<tr>
<td>1</td>
<td>logging.INFO</td>
</tr>
<tr>
<td>2</td>
<td>logging.DEBUG</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()`

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)
```

```python
def set_verbose_files(file_name):
    pass

def unset_verbose_files(file_name):
    pass

def 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
VERBOSE1 (william): This is Sage. (time = 0.0)
sage: set_verbose(0)
```

### 2.8 Laziness

#### 2.8.1 Lazy attributes

AUTHORS:

• Nicolas Thiery (2008): Initial version
• Nils Bruin (2013-05): Cython version

```python
class sage.misc.lazy_attribute.lazy_attribute(f):
    Bases: sage.misc.lazy_attribute._lazy_attribute
```
A lazy attribute for an object is like a usual attribute, except that, instead of being computed when the object is constructed (i.e. in `__init__`), it is computed on the fly the first time it is accessed.

For constant values attached to an object, lazy attributes provide a shorter syntax and automatic caching (unlike methods), while playing well with inheritance (like methods): a subclass can easily override a given attribute; you don’t need to call the super class constructor, etc.

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

**EXAMPLES:**

We create a class whose instances have a lazy attribute `x`:

```python
sage: class A(object):
....:     def __init__(self):
....:         self.a=2 # just to have some data to calculate from
....:
....:     @lazy_attribute
....:     def x(self):
....:         print("calculating x in A")
....:         return self.a + 1
....:
```

For an instance `a` of `A`, `a.x` is calculated the first time it is accessed, and then stored as a usual attribute:

```python
sage: a = A()
... calculating x in A
sage: a.x
3
sage: a.x
3
```

**Implementation details**

We redo the same example, but opening the hood to see what happens to the internal dictionary of the object:

```python
sage: a = A()
... calculating x in A
sage: a.__dict__
{'a': 2}
sage: a.x
3
sage: a.__dict__
{'a': 2, 'x': 3}
sage: a.x
3
sage: timeit('a.x')   # random
625 loops, best of 3: 89.6 ns per loop
```

This shows that, after the first calculation, the attribute `x` becomes a usual attribute; in particular, there is no time penalty to access it.

A lazy attribute may be set as usual, even before its first access, in which case the lazy calculation is completely ignored:
Class binding results in the lazy attribute itself:

```python
sage: A.x
<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
3
sage: b = B()
sage: b.y = 1
sage: b.x
calculating x from y in B
1
```

Attribute existence testing

Testing for the existence of an attribute with `hasattr` currently always triggers its full calculation, which may not be desirable when the calculation is expensive:

```python
sage: a = A()
sage: hasattr(a, "x")
calculating x in A
True
```

It would be great if we could take over the control somehow, if at all possible without a special implementation of `hasattr`, so as to allow for something like:
sage: class A (object):
....:     @lazy_attribute
....:     def x(self, existence_only=False):
....:         if existence_only:
....:             print("testing for x existence")
....:             return True
....:         else:
....:             print("calculating x in A")
....:             return 3

sage: a = A()
sage: hasattr(a, "x")  # todo: not implemented
testing for x existence
sage: a.x
calculating x in A
3
sage: a.x
3

Here is a full featured example, with both conditional definition and existence testing:

sage: class B(A):
....:     @lazy_attribute
....:     def x(self, existence_only=False):
....:         if hasattr(self, "y"):
....:             if existence_only:
....:                 print("testing for x existence in B")
....:                 return True
....:             else:
....:                 print("calculating x from y in B")
....:                 return self.y
....:         else:
....:             print("y not there; B does not define x")
....:             return NotImplemented

sage: b = B()
sage: hasattr(b, "x")  # todo: not implemented
y not there; B does not define x
testing for x existence
True
sage: b.x
y not there; B does not define x
calculating x in A
3
sage: b = B()
sage: b.y = 1
sage: hasattr(b, "x")  # todo: not implemented
testing for x existence in B
True
sage: b.x
calculating x from y in B
1
lazy attributes and introspection

Todo: Make the following work nicely:

```python
sage: b.x? # todo: not implemented
sage: b.x?? # todo: not implemented
```

Right now, the first one includes the doc of this class, and the second one brings up the code of this class, both being not very useful.

Partial support for old style classes

Old style and new style classes play a bit differently with @property and attribute setting:

```python
sage: class A: # py2 - no old-style classes on python 3
    ....: @property
    ....: def x(self):
    ....:     print("calculating x")
    ....:     return 3
    ....:
    sage: a = A() # py2
    sage: a.x = 4 # py2
    sage: a.__dict__ # py2
    {'x': 4}
    sage: a.x # py2
    4
    sage: a.__dict__['x']=5 # py2
    sage: a.x # py2
    5

sage: class A (object):
    ....: @property
    ....: def x(self):
    ....:     print("calculating x")
    ....:     return 3
    ....:
    sage: a = A()
    sage: a.x = 4
    Traceback (most recent call last):
    ...
    AttributeError: can't set attribute
    sage: a.__dict__
    {}
    sage: a.x
    calculating x
    3
    sage: a.__dict__['x']=5
    sage: a.x
    calculating x
    3
```
In particular, lazy_attributes need to be implemented as non-data descriptors for new style classes, so as to leave access to setattr. We now check that this implementation also works for old style classes (conditional definition does not work yet):

```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")
....:         return self.a + 1
sage: a = A()
sage: a.__dict__
{'a': 2}
sage: a.x
calculating x
3
sage: a.__dict__
{'a': 2, 'x': 3}
sage: a.x
3
sage: timeit('a.x')  # random
625 loops, best of 3: 115 ns per loop
sage: a = A()
sage: a.x = 4
sage: a.x
4
sage: a.__dict__
{'a': 2, 'x': 4}
```

```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  # todo: not implemented
y not there; B does not define x
calculating x in A
3
sage: b = B()
sage: b.y = 1
sage: b.x
calculating x from y in B
1
```
Lazy attributes and Cython

This attempts to check that lazy attributes work with built-in functions like cpdef methods:

```python
sage: class A:
    ....:     def __len__(x):
    ....:         return int(5)
    ....:     len = lazy_attribute(len)

sage: A().len
5
```

Since trac ticket #11115, extension classes derived from Parent can inherit a lazy attribute, such as element_class:

```python
sage: cython_code = ['from sage.structure.parent cimport Parent',
                    'from sage.structure.element cimport Element',
                    'cdef class MyElement(Element): pass',
                    'cdef class MyParent(Parent):',
                    '    Element = MyElement']

sage: cython('
'.join(cython_code))

sage: P = MyParent(category=Rings())

sage: P.element_class
# indirect doctest
<type '...MyElement'>
```

About descriptor specifications

The specifications of descriptors (see 3.4.2.3 Invoking Descriptors in the Python reference manual) are incomplete w.r.t. inheritance, and maybe even ill-implemented. We illustrate this on a simple class hierarchy, with an instrumented descriptor:

```python
sage: class descriptor(object):
    ....:     def __get__(self, obj, cls):
    ....:         print(cls)
    ....:         return 1

sage: class A(object):
    ....:     x = descriptor()

sage: class B(A):
    ....:         pass

sage: A.x
<class '__main__.A'>
1

sage: B().x
<class '__main__.B'>
1
```

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:

```python
sage: B().x
<class '__main__.B'>
1
```
It would seem more natural (and practical!) to get A rather than B.

From the specifications for Super Binding, it would be expected to get A and not B as cls parameter:

```
sage: super(B, B()).x
<class '__main__.B'>
1
```

Due to this, the natural implementation runs into an infinite loop in the following example:

```
sage: class A(object):
    ....:     @lazy_attribute
    ....:     def unimplemented_A(self):
    ....:         return NotImplemented
    ....:     @lazy_attribute
    ....:     def unimplemented_AB(self):
    ....:         return NotImplemented
    ....:     @lazy_attribute
    ....:     def unimplemented_B_implemented_A(self):
    ....:         return 1
    ....:
sage: class B(A):
    ....:     @lazy_attribute
    ....:     def unimplemented_B(self):
    ....:         return NotImplemented
    ....:     @lazy_attribute
    ....:     def unimplemented_AB(self):
    ....:         return NotImplemented
    ....:     @lazy_attribute
    ....:     def unimplemented_B_implemented_A(self):
    ....:         return NotImplemented
    ....:
sage: class C(B):
    ....:     pass
    ....:
```

This is the simplest case where, without workaround, we get an infinite loop:

```
sage: hasattr(B(), "unimplemented_A") # todo: not implemented
False
```

**Todo:** Improve the error message:

```
sage: B().unimplemented_A # todo: not implemented
Traceback (most recent call last):
...
AttributeError: 'super' object has no attribute 'unimplemented_A'
```

We now make some systematic checks:

```
sage: B().unimplemented_A
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_A'
sage: B().unimplemented_B
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_B'
sage: B().unimplemented_AB
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_AB'
sage: B().unimplemented_B_implemented_A
1

sage: C().unimplemented_A()
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_A'
sage: C().unimplemented_B()
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_B'
sage: C().unimplemented_AB()
Traceback (most recent call last):
...
AttributeError: '...' object has no attribute 'unimplemented_AB'
sage: C().unimplemented_B_implemented_A  # todo: not implemented
1

class sage.misc.lazy_attribute.lazy_class_attribute(f)
Bases: sage.misc.lazy_attribute.lazy_attribute

A lazy class attribute for an class is like a usual class attribute, except that, instead of being computed when the class is constructed, it is computed on the fly the first time it is accessed, either through the class itself or through one of its objects.

This is very similar to lazy_attribute except that the attribute is a class attribute. More precisely, once computed, the lazy class attribute is stored in the class rather than in the object. The lazy class attribute is only computed once for all the objects:

```python
sage: class Cl(object):
    ....:     @lazy_class_attribute
    ....:     def x(cls):
    ....:         print("computing x")
    ....:         return 1
sage: Cl.x
computing 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
```
First access from an object also properly triggers the computation:

```
sage: class Cl1(object):
....:     @lazy_class_attribute
....:     def x(cls):
....:         print("computing x")
....:         return 1
sage: Cl1().x
computing x
1
```

**Warning:** The behavior of lazy class attributes with respect to inheritance is not specified. It currently depends on the evaluation order:

```
sage: class A(object):
....:     @lazy_class_attribute
....:     def x(cls):
....:         print("computing x")
....:         return str(cls)
....:     @lazy_class_attribute
....:     def y(cls):
....:         print("computing y")
....:         return str(cls)
sage: class B(A):
....:     pass

sage: A.x
computing x
"<class '__main__.A'>"
sage: B.x
"<class '__main__.A'>"

sage: B.y
computing y
"<class '__main__.B'>"
sage: A.y
computing y
"<class '__main__.A'>"
sage: B.y
"<class '__main__.B'>"
```
2.8.2 Lazy format strings

```python
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(object):
    ....:     def __repr__(self):
    ....:         raise ValueError("Don't ever try to print me !")

There is no error when binding a lazy format with the broken object:

```python
sage: lf = LazyFormat("<%s>")%IDontLikeBeingPrinted()
```

The error only occurs upon printing:

```python
sage: lf
<repr(<sage.misc.lazy_format.LazyFormat at 0x...>) failed: ValueError: Don't ever try to print me !>
```

**Common use case:**

Most of the time, `__repr__` methods are only called during user interaction, and therefore need not be fast; and indeed there are objects `x` in Sage such `x.__repr__()` is time consuming.

There are however some uses cases where many format strings are constructed but not actually printed. This includes error handling messages in `unittest` or `TestSuite` executions:

```python
sage: QQ._tester().assertTrue(0 in QQ,
    ....:     "%s doesn't contain 0"%QQ)
```

In the above `QQ.__repr__()` has been called, and the result immediately discarded. To demonstrate this we replace `QQ` in the format string argument with our broken object:
sage: QQ._tester().assertTrue(True,
....:       "%s doesn't contain 0"%IDontLikeBeingPrinted())
Traceback (most recent call last):
...  
  ValueError: Don't ever try to print me!

This behavior can induce major performance penalties when testing. Note that this issue does not impact the usual assert:

sage: assert True, "%s is wrong"%IDontLikeBeingPrinted()

We now check that LazyFormat indeed solves the assertion problem:

sage: QQ._tester().assertTrue(True,
....:       LazyFormat("%s is wrong")%IDontLikeBeingPrinted())
sage: QQ._tester().assertTrue(False,
....:       LazyFormat("%s is wrong")%IDontLikeBeingPrinted())
Traceback (most recent call last):
...  
  AssertionError: <unprintable AssertionError object>

## 2.8.3 Lazy imports

This module allows one to lazily import objects into a namespace, where the actual import is delayed until the object is actually called or inspected. This is useful for modules that are expensive to import or may cause circular references, though there is some overhead in its use.

**EXAMPLES:**

sage: lazy_import('sage.rings.all', 'ZZ')
sage: type(ZZ)
<type 'sage.misc.lazy_import.LazyImport'>
sage: ZZ(4.0)
4

By default, a warning is issued if a lazy import module is resolved during Sage's startup. In case a lazy import's sole purpose is to break a circular reference and it is known to be resolved at startup time, one can use the `at_startup` option:

sage: lazy_import('sage.rings.all', 'ZZ', at_startup=True)

This option can also be used as an intermediate step toward not importing by default a module that is used in several places, some of which can already afford to lazy import the module but not all.

A lazy import that is marked as "at_startup" will print a message if it is actually resolved after the startup, so that the developer knows that (s)he can remove the flag:

sage: ZZ
Option `\`at_startup=True\` for lazy import ZZ not needed anymore
Integer Ring

**See also:**

`lazy_import()`, `LazyImport`

### 2.8. Laziness
AUTHOR:

- Robert Bradshaw

class sage.misc.lazy_import.LazyImport

    Bases: object

    EXAMPLES:

```
sage: from sage.misc.lazy_import import LazyImport
sage: my_integer = LazyImport('sage.rings.all', 'Integer')
sage: my_integer(4)
4
sage: my_integer('101', base=2)
5
sage: my_integer(3/2)
Traceback (most recent call last):
...  
TypeError: no conversion of this rational to integer
```

sage.misc.lazy_import.finish_startup()

This function must be called exactly once at the end of the Sage import process.

sage.misc.lazy_import.get_star_imports(module_name)

Lookup the list of names in a module that would be imported with “import *” either via a cache or actually importing.

    EXAMPLES:

```
sage: from sage.misc.lazy_import import get_star_imports
sage: 'get_star_imports' in get_star_imports('sage.misc.lazy_import')
True
sage: 'EllipticCurve' in get_star_imports('sage.schemes.all')
True
```

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
• deprecation – (optional) if not None, a deprecation warning will be issued when the object is actually imported; deprecation should be either a trac number (integer) or a pair (trac_number, message)
• feature – a python module (optional), if it cannot be imported an appropriate error is raised

See also:
sage.misc.lazy_import, LazyImport

EXAMPLES:

```
sage: lazy_import('sage.rings.all', 'ZZ')

sage: type(ZZ)
<type 'sage.misc.lazy_import.LazyImport'>

sage: ZZ(4.0)
4

sage: lazy_import('sage.rings.all', 'RDF', 'my_RDF')

sage: my_RDF._get_object() is RDF
True

sage: my_RDF(1/2)
0.5

sage: lazy_import('sage.all', [QQ', ', 'RR'], ['my_QQ', ', 'my_RR'])

sage: my_QQ._get_object() is QQ
True

sage: my_RR._get_object() is RR
True
```

Upon the first use, the object is injected directly into the calling namespace:

```
sage: lazy_import('sage.all', 'ZZ', 'my_ZZ')

sage: my_ZZ is ZZ
False

sage: my_ZZ(37)
37

sage: my_ZZ is ZZ
True
```

We check that `lazy_import()` also works for methods:

```
sage: class Foo(object):
    ....:    lazy_import('sage.all', 'plot')

sage: class Bar(Foo):
    ....:    pass

sage: type(Foo.__dict__['plot'])
<type 'sage.misc.lazy_import.LazyImport'>

sage: 'EXAMPLES' in Bar.plot.__doc__
True

sage: type(Foo.__dict__['plot'])
<... 'function'>
```

If deprecated then a deprecation warning is issued:
An example of deprecation with a message:

```python
sage: lazy_import('sage.all', 'Qp', 'my_Qp', deprecation=(14275, "This is an example."))
sage: my_Qp_msg(5)
```

```python
doctest:...: DeprecationWarning: This is an example.
See http://trac.sagemath.org/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'))
sage: equation
<built-in function equation>
```

```python
sage: lazy_import('PyNormaliz', 'NmzListConeProperties', feature=PythonModule('PyNormaliz', spkg='pynormaliz')) # optional - pynormaliz
sage: NmzListConeProperties # optional - pynormaliz
<built-in function NmzListConeProperties>
```

```python
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...
```

---

`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:

```python
sage: sage.misc.lazy_import.test_fake_startup()
sage: lazy_import('sage.rings.all', 'ZZ', 'my_ZZ')
sage: my_ZZ(123)
Resolving lazy import ZZ during startup
123
sage: sage.misc.lazy_import.finish_startup()
```
### 2.8.4 Lazy import cache

This is a pure Python file with no dependencies so it can be used in setup.py.

```python
sage.misc.lazy_import_cache.get_cache_file()
```

Return the canonical filename for caching names of lazily imported modules.

**EXAMPLES:**

```python
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
```

It should not matter whether DOT_SAGE ends with a slash:

```python
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
```

### 2.8.5 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:**

```python
sage: from sage.misc.lazy_list import lazy_list
sage: P = lazy_list(Primes())

sage: P[100]
547

sage: P[10:34]
lazy list [31, 37, 41, ...]

sage: P[12:23].list()
[41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83]

sage: f = lazy_list((i**2 - 3*i for i in 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)]
[0, -2]

sage: [next(i2), next(i2)]
[0, -2]

sage: [next(i1), next(i1)]
[-2, -2]
```

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

```python
2.8. Laziness  159
```
```python
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:

```python
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
```

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 $1 \rightarrow 10$:

```python
sage: from sage.misc.lazy_list import lazy_list_generic
sage: class MyThueMorseWord(lazy_list_generic):
    ....:    def __init__(self):
    ....:        self.i = 1
    ....:        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:

```python
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
```

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`. 

160 Chapter 2. Programming Utilities
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.

```sage
from sage.misc.lazy_list import lazy_list
```

1. Iterators:

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

2. Functions:

```sage
lazy_list(lambda n: (n**2)%17)
lazy list [0, 1, 4, ...]
```

3. Plain lists:

```sage
lazy_list([1,5,7,2])
lazy list [1, 5, 7, ...]
```

If a function is only defined for large values, you can provide the beginning of the sequence manually:

```sage
l = lazy_list(divisors, [None])
l
lazy list [None, [1], [1, 2], ...]
```

Lazy lists behave like lists except that they are immutable:

```sage
l[3::5]
lazy list [[1, 3], [1, 2, 4, 8], [1, 13], ...]
```
If your lazy list is finite, you can obtain the underlying list with the method `.list()`:

```
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]]
```

```
sage.misc.lazy_list.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:**

```
sage: from sage.misc.lazy_list import lazy_list_formatter
sage: lazy_list_formatter(srange(3, 1000, 5), name='list')
'list [3, 8, 13, ...]'
sage: from sage.misc.lazy_list import lazy_list
sage: L = lazy_list(Primes()); L
lazy list [2, 3, 5, ...]
sage: repr(L) == lazy_list_formatter(L)
True
sage: lazy_list_formatter(L, name='primes')
'primes [2, 3, 5, ...]'
sage: lazy_list_formatter(L, opening_delimiter='(', closing_delimiter=')')
'lazy list (2, 3, 5, ...)'
sage: lazy_list_formatter(L, opening_delimiter=' ', closing_delimiter='')
'lazy list 2, 3, 5, ...'
sage: lazy_list_formatter(L, separator='--')
'lazy list [2--3--5--...]'sage: lazy_list_formatter(L, more='and more')
'lazy list [2, 3, 5, and more]'
sage: lazy_list_formatter(L, preview=10)
'lazy list [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ...]'
sage: lazy_list_formatter(L, name='primes',
   opening_delimiter=' ', closing_delimiter=''
)
(s continues on next page)
class sage.misc.lazy_list.lazy_list_from_function
Bases: sage.misc.lazy_list.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)
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: sage.misc.lazy_list.lazy_list_generic

Lazy list built from an iterator.

EXAMPLES:

```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]
lazy list [8, 10, 12, ...]
sage: x = iter(m)
sage: [next(x), next(x), next(x)]
[0, 1, 2]
```

class sage.misc.lazy_list.lazy_list_from_update_function
Bases: sage.misc.lazy_list.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

2.8. Laziness
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)

```python
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
    lazy list [2, 3, 5, ...]
    sage: l[200]
    1229
```

### 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`.

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

sage: g = f[:10]
sage: g.get(5)
'a'

sage: g.get(10)
Traceback (most recent call last):
  ...  
IndexError: lazy list index out of range

sage: g.get(1/2)
Traceback (most recent call last):
  ...  
TypeError: unable to convert rational 1/2 to an integer
```

### 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:

```python
sage: from sage.misc.lazy_list import lazy_list
sage: P = lazy_list(Primes())
sage: P[2:143:5].list()
   457, 487, 521, 563, 593, 617, 647, 677, 719, 751, 787, 823]
sage: P = lazy_list(iter([1,2,3]))
sage: P.list()
[1, 2, 3]
sage: P[:100000].list()
[1, 2, 3]
sage: P[1:7:2].list()
[2]
```

`sage.misc.lazy_list.slice_unpickle(master, start, stop, step)`
Unpickle helper

### 2.8.6 Lazy strings

Based on 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. Sage uses lazy strings in `sage.misc.misc` so that the filenames for SAGE_TMP (which depends on the pid of the process running Sage) are not computed when importing the Sage library. This means that when the doctesting code imports the Sage library and then forks, the variable SAGE_TMP depends on the new pid rather than the old one.

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
l'1'
```

Note that the function is recomputed each time:

```python
sage: L.append(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
```
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
l' 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
sage: unicode(s)  # py2
determining string representation
u' this is a test'
```

## 2.9 Caching

### 2.9.1 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 trac ticket #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
```

In some cases, one would only want to keep the result in cache as long as there is any other reference to the result. By trac ticket #12215, this is enabled for UniqueRepresentation, which is used to create unique parents: If an algebraic structure, such as a finite field, is only temporarily used, then it will not stay in cache forever. That behaviour is implemented using weak_cached_function, that behaves the same as cached_function, except that it uses a 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 trac ticket #11115. We need to provide the name that the wrapped method or function should have, since otherwise the name of the original function would be used:

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

sage: wrapped_funct.__name__
'wrapped_funct'
sage: wrapped_funct(5)
-5
sage: wrapped_funct(5) is wrapped_funct(5)
True
```

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

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

(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 trac ticket #12215, this is enabled for UniqueRepresentation, which is used to create unique parents: If an algebraic structure, such as a finite field, is only temporarily used, then it will not stay in cache forever. That behaviour is implemented using weak CachedFunction, that behaves the same as cached function, except that it uses a CachedWeakValueDictionary for storing the results.

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

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.
The cached methods inherited by the parent works:

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

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

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

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

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

The cache can be emptied:

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

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

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

Introspection works:

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

(continues on next page)
Some doc for direct method

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

sage: print(sage_getsource(O.direct_method))
def direct_method(self, x):
    "Some doc for direct method"
    return 2*x
```

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

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

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

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

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

```python
sage: cython_code = [
    ....: "from sage.misc.cachefunc import cached_method",
    ....: "cdef class MyOtherClass:",
    ....: "    cdef dict D",
    ....: "    def __init__(self):
        self.D = {}",
    ....: "    def __setattr__(self, n,v):
        self.D[n] = v",
    ....: "    def __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"
]
```

```python
sage: cython(os.linesep.join(cython_code))
```

```python
sage: Q = MyOtherClass()
```

```python
sage: Q.f(2,3)
5
```

```python
sage: Q.f(2,3) is Q.f(2,3)  
True
```

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

```python
sage: timeit("a = P.f(2,3)")  
625 loops, best of 3: 1.3 µs per loop
```

```python
sage: timeit("a = Q.f(2,3)")  
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: K.<a> = Qq(9)
sage: b = a.add_bigoh(1)
sage: c = a + 3
```

```python
sage: b
a + O(3)
sage: c
a + 3 + O(3^20)
sage: b == c  
True
```

```python
sage: b == a  
True
```

```python
sage: c == a  
False
```

If such objects defined a non-trivial hash function, this would break caching in many places. However, such objects should still be usable in caches. This can be achieved by defining an appropriate method `_cache_key`: 
```python
sage: hash(b)
Traceback (most recent call last):
...  
TypeError: unhashable type: 'sage.rings.padics.qadic_flint_CR.qAdicCappedRelativeElement'
```

```python
sage: @cached_method
def f(x): return x == a
sage: f(b)
True
sage: f(c)  # if b and c were hashable, this would return True
False
```

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

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

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

```python
sage: S.<a> = Qq(4)
sage: d = a.add_bigoh(1)
sage: b._cache_key() == d._cache_key()  # this would be True if the parents were not included
False
```

```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 `sage.structure.sage_object.SageObject._cache_key()`.

**EXAMPLES:**

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

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

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

The user can clear the cache:

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

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

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

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

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

```
is mul(1,1,algorithm="algorithm")
is mul(1,1)
is...
```

```
True
```

**cache**

**cached(**`*args`, **`*kwd`s]**

Return the result from the cache if available. If the value is not cached, raise `KeyError`.

**EXAMPLES:**

```
sage: @cached_function
def f(x):
    return x
sage: f.cached(5)
Traceback (most recent call last):
...
KeyError: ((5,), ())
```

(continues on next page)
clear_cache()

Clear the cache dictionary.

EXAMPLES:

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

f

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,), ())
```

Examples for cached methods:

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

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

```
sage: a.f.get_key(1, 2, 3, x=4, y=5, z=6)
((1, 2, 3), ('x', 4), ('y', 5), ('z', 6))
```
It does, however, take into account the different ways of providing named arguments, possibly with a default value:

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

`is_in_cache(*args, **kwds)`
Checks if the argument list is in the cache.

**EXAMPLES:**

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

`precompute(arglist, num_processes=1)`
Cache values for a number of inputs. Do the computation in parallel, and only bother to compute values that we haven’t already cached.

**INPUT:**

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

**EXAMPLES:**

```
sage: @cached_function
    ....:     def oddprime_factors(n):
    ....:         l = [p for p,e in factor(n) if p != 2]
    ....:         return len(l)
sage: oddprime_factors.precompute(range(1, 100), 4)
sage: oddprime_factors.cache[(25,), ()]  # cell 1
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: 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
class sage.misc.cachefunc.CachedInParentMethod
Bases: sage.misc.cachefunc.CachedMethod

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

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

**ASSUMPTION:**

This way of caching works only if

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

**NOTE:**

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

Examples can be found at `cache.func`.

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

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

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

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

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

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

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

sage: a = A()
sage: a.f(1, algorithm="default")
```

The parameter `do_pickle` can be used to enable pickling of the cache. Usually the cache is not stored when pickling:

```
sage: class A(object):
    ....:     @cached_method
    ....:     def f(self, x):
    ....:         return None

sage: import __main__
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)
0
```

When `do_pickle` is set, the pickle contains the contents of the cache:

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

(continues on next page)
Cached methods cannot be copied like usual methods, see trac ticket #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):
      ....:       return 2
sage: b=B()
sage: b.f()
2
sage: b.g()
1
sage: b.f()
1
```

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

Bases: `sage.misc.cachefunc.CachedFunction`

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

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

**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)
<type 'sage.misc.cachefunc.CachedMethodCaller'>
sage: a.bar(2) is a.bar(x=2)
True
```

**cached(**`*args, **kwsargs`**)**

Return the result from the cache if available. If the value is not cached, raise `KeyError`.

**EXAMPLES:**

2.9. Caching
```python
sage: class CachedMethodTest(object):
    ....:    @cached_method
    ....:    def f(self, x):
    ....:        return x
sage: o = CachedMethodTest()
sage: CachedMethodTest.f.cached(o, 5)
Traceback (most recent call last):
  ... KeyError: ((5,), ())
sage: o.f.cached(5)
Traceback (most recent call last):
  ... KeyError: ((5,), ())
sage: o.f(5)
5
sage: CachedMethodTest.f.cached(o, 5)
5
sage: 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(object):
    ....:    @cached_method
    ....:    def f(self, i):
    ....:        return i^2
sage: foo = Foo()
sage: foo.f(3)
9
sage: foo.f(1)
1
sage: foo.f.precompute(range(2), 2)
sage: foo.f.cache == {((0,), ()): 0, ((1,), ()): 1, ((3,), ()): 9}
True
```

```python
class sage.misc.cachefunc.CachedMethodCallerNoArgs
    Bases: sage.misc.cachefunc.CachedFunction

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

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

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

AUTHOR:
  - Simon King (2011-04)

**clear_cache()**
Clear the cache dictionary.

**EXAMPLES:**

\begin{verbatim}
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'
\end{verbatim}

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

\begin{verbatim}
sage: I.gens.clear_cache()
sage: I.gens()
[a, b]
\end{verbatim}

**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:**

\begin{verbatim}
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
\end{verbatim}

**set_cache(value)**
Override the cache with a specific value.

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

EXAMPLES:

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

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

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

The attempt to assign None to the cache fails:

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

```
class sage.misc.cachefunc.CachedMethodPickle(inst, name, cache=None)
Bases: object

This class helps to unpickle cached methods.

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

EXAMPLES:

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

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

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

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

• Simon King (2011-01)

class sage.misc.cachefunc.CachedSpecialMethod

Bases: sage.misc.cachefunc.CachedMethod

Cached version of special python methods.

IMPLEMENTATION:

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

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

Here, the \_\_get\__ method will explicitly access the \_\_dict\_, so that hash(c) will rely on a single CachedMethodCaller stored in the \_\_dict\_.

EXAMPLES:

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

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

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

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

Bases: sage.misc.cachefunc.CachedFunction

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

EXAMPLES:
```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
```

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 `memory_cache == True`, we maintain a cache of objects seen during this session in memory – but we don’t load them from disk until necessary. The keys and values are stored in a pair of files:

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

where `self[key] == val`.

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

clear()

Clear all key, value pairs from `self` and unlink the associated files from the file cache.

**EXAMPLES:**

```python
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: k1 in FC1
False
sage: k2 in FC1
False
```

file_list()

Return the list of files corresponding to `self`.

**EXAMPLES:**
sage: from sage.misc.cachefunc import FileCache
sage: dir = tmp_dir()

sage: FC = FileCache(dir, memory_cache = True, prefix='t')

sage: FC[((0,0))] = 1
sage: FC[((1,2),0)] = 2
sage: FC[((1,),('a',1,)))] = 3

sage: for f in sorted(FC.file_list()): print(f[len(dir):])
t-.key.sobj
t-.sobj
t-1_2.key.sobj
t-1_2.sobj
t-a-1.1.key.sobj
t-a-1.1.sobj

**items()**

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

EXAMPLES:

```python
sage: from sage.misc.cachefunc import FileCache
sage: dir = tmp_dir()

sage: FC = FileCache(dir, memory_cache = False)

sage: FC[((0,0))] = 1
sage: FC[((1,2),0)] = 2
sage: FC[((1,),('a',1,)))] = 3

sage: I = FC.items()
sage: 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
sage: dir = tmp_dir()

sage: FC = FileCache(dir, memory_cache = False)

sage: FC[((0,0))] = 1
sage: FC[((1,2),0)] = 2
sage: FC[((1,),('a',1,)))] = 3

sage: K = FC.keys()
sage: K.sort(); K
[(((), ()), (1,1), ((('a', 1),)), 3), (((1, 2), ()), 2)]
```

**values()**

Return a list of values that are stored in \(self\).

EXAMPLES:

```python
sage: from sage.misc.cachefunc import FileCache
sage: dir = tmp_dir()

sage: FC = FileCache(dir, memory_cache = False)

sage: FC[((0,0))] = 1
sage: FC[((1,2),0)] = 2
sage: FC[((1,),('a',1,)))] = 3
```

(continues on next page)
class sage.misc.cachefunc.GloballyCachedMethodCaller
   Bases: sage.misc.cachefunc.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()
   sage: d[0] = 0
   sage: loads(dumps(d))
   {}

class sage.misc.cachefunc.WeakCachedFunction
   Bases: sage.misc.cachefunc.CachedFunction

   A version of CachedFunction using weak references on the values.

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

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

   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:

   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
dsage: del b
da = f()
```

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()
da = f(1); del a
da = f(2), f(1); del a
da = f(3), f(1); del a
da = f(4), f(1); del a
da = f(5), f(1); del a
```

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

```
sage: @weak_cached_function(key=lambda x,algorithm: x)
da = f(1,algorithm="default")
da = f(1,algorithm="algorithm")
```

This function works if `o` is a tuple. In this case it unpacks its entries recursively:

```
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:
```
sage: from sage.misc.cachefunc import cache_key
sage: K.<u> = Qq(9)
sage: a = K(1); a
1 + O(3^20)
sage: cache_key(a)
(..., ((1,),), 0, 20)
```

This function works if `o` is a tuple. In this case it unpacks its entries recursively:
sage: o = (1, 2, (3, a))
sage: cache_key(o)
(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)
sage: cache_key(o)
(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
• 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):

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

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

EXAMPLES:

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

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

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

The user can clear the cache:

```
sage: a = mul(4)
sage: mul.clear_cache()
sage: a is mul(4)
False
```
It is also possible to explicitly override the cache with a different value:

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

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

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

### sage.misc.cachefunc.cached_in_parent_method

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

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

**ASSUMPTION:**

This way of caching works only if

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

**NOTE:**

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

Examples can be found at `cachefunc`.

### sage.misc.cachefunc.cached_method

A decorator for cached methods.

**EXAMPLES:**

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

```
sage: class C:
    ....:     @cached_method
    ....:     def __hash__(self):
    ....:         print("compute hash")
    ....:         return int(5)
    ....:     @cached_method
    ....:     def f(self, x):
    ....:         print("computing cached method")
    ....:         return x^2
sage: c = C()
sage: type(C.__hash__)
```

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

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

Different instances have distinct caches:

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

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

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

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__`:

```
sage: class C:
....:     @cached_method(do_pickle=True)
....:     def __hash__(self):
....:         return id(self)
sage: import __main__
sage: __main__.C = C
sage: c = C()  # random output
sage: hash(c)  # random output
sage: d = loads(dumps(c))
sage: hash(d) == hash(c)  # True
```

However, the contents of a method’s cache are not pickled unless `do_pickle` is set:
```python
sage: class C:
    ....:     @cached_method
    ....:     def __hash__(self):
    ....:         return id(self)

sage: __main__.C = C
sage: c = C()
sage: hash(c)  # random output
sage: d = loads(dumps(c))
sage: 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:**

```python
sage: from sage.misc.cachefunc import dict_key
sage: dict_key(42)
42
sage: K.<u> = Qq(9)
sage: dict_key(u)
(<object object at ...>, (..., 20))
```

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

Bases: object

Decorator for DiskCachedFunction.

**EXAMPLES:**

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

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

A version of CachedFunction using weak references on the values.

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

```
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
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
doing a computation for x=1
sage: a = f(3), f(1); del a
doing a computation for x=3
doing a computation for x=2
sage: a = f(4), f(1); del a
doing a computation for x=4
doing a computation for x=3
sage: a = f(5), f(1); del a
doing a computation for x=5
doing a computation for x=4
```

The parameter `key` can be used to ignore parameters for caching. In this example we ignore the parameter `algorithm`:
2.9.2 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:

```python
sage: import weakref
sage: class Vals(object):
    pass
sage: class Keys:
        def __init__(self, val):
            self.val = weakref.ref(val)
        def __hash__(self):
            return hash(self.val())
        def __eq__(self, other):
            return self.val() == other.val()
        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 occurs as value for several keys, each reference to it corresponds to a unique key. We see no error messages, and the items get correctly removed:

```python
sage: ValsList = [Vals() for _ in range(10)]
sage: import sage.misc.weak_dict
```

(continues on next page)
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 trac ticket #13394 for a discussion of some of the design considerations.

```python
class sage.misc.weak_dict.CachedWeakValueDictionary
    Bases: sage.misc.weak_dict.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(object): pass
 sage: tmp = Test()
 sage: D[0] = tmp
 sage: 0 in D
 True
 sage: del tmp
 sage: 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(object): 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
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:
```
```
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:

```python
sage: import weakref
sage: class Vals(object):
... pass
sage: class Keys:
...     def __init__(self, val):
...         self.val = weakref.ref(val)
...     def __hash__(self):
...         return hash(self.val())
...     def __eq__(self, other):
...         return self.val() == other.val()
...     def __ne__(self, other):
...         return self.val() != other.val()

sage: ValList = [Vals() for __ in range(10)]

sage: import sage.misc.weak_dict
sage: D = sage.misc.weak_dict.WeakValueDictionary()

sage: for v in ValList:
...     D[Keys(v)] = v

sage: len(D)
10

sage: del ValList
sage: len(D)
1

sage: del v
sage: len(D)
0

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:

```python
sage: import sage.misc.weak_dict
sage: L = [GF(p) for p in prime_range(10^3)]

sage: D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))

sage: 100 in D
True
```

(continues on next page)
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

**items()**
The key-value pairs of this dictionary.

**EXAMPLES:**

```python
type(3)
type(3)
NotImplemented
type(3)
```

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

```python
Sage: del D[Keys(2)]
Sage: del L[5]
Sage: sorted(D.items())
[((0), <0>),
 (1), <1>),
 (3), <3>),
```
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 "<{}>".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()

sage: class Keys(object):
    ....:     def __init__(self, n):
    ....:         self.n = n
    ....:     def __hash__(self):
    ....:         if self.n % 2:
    ....:             return hash(5)
    ....:         else:
    ....:             return hash(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()

sage: L = [(Keys(n), Vals(n)) for n in range(10)]

sage: D = sage.misc.weak_dict.WeakValueDictionary(L)

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

```python
sage: del D[Keys(2)]
sage: del L[5]
sage: for k,v in sorted(D.iteritems()):
    ....:     print("{} {}":format(k, v))
```

(continues on next page)
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()

sage: L = [Vals(n) for n in range(10)]
sage: D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))

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

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

The list of keys.

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

sage: D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))

sage: del L[4]

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

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

pop(k)
Return the value for a given key, and delete it from the dictionary.

EXAMPLES:

sage: import sage.misc.weak_dict
sage: L = [GF(p) for p in prime_range(10^3)]

sage: D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))

sage: 20 in D
True

sage: D.pop(20)
Finite Field of size 73

sage: 20 in D
False

sage: D.pop(20)
Traceback (most recent call last):
  ... KeyError: 20

popitem()
Return and delete some item from the dictionary.

EXAMPLES:

sage: import sage.misc.weak_dict
sage: D = sage.misc.weak_dict.WeakValueDictionary()

sage: D[1] = ZZ

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

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

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

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

setdefault(k, default=None)
Return the stored value for a given key; return and store a default value if no previous value is stored.

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

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

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

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

values()

Return the list of values.

EXAMPLES:

```
sage: import sage.misc.weak_dict
sage: class Vals:
    ....:     def __init__(self, n):
    ....:         self.n = n
    ....:     def __repr__(self):
    ....:         return "<%s>" % self.n
    ....:     def __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 = [Vals(n) for n in range(10)]
```

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

```
sage: del D[2]
sage: del L[5]
sage: sorted(D.values())
[<0>, <1>, <3>, <4>, <6>, <7>, <8>, <9>]
```
2.10 Fast Expression Evaluation

2.10.1 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, fast_callable(), to transform such expressions into a form where they can be evaluated quickly:

```
sage: f = sin(x) + 3*x^2
sage: ff = fast_callable(f, vars=[x])
sage: ff(3.5)
36.3992167723104
sage: ff(RIF(3.5))
36.39921677231038?
```

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:

```
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=

```
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*x^2 with domain=D would be equivalent to D(D(sin(D(x)))) + D(D(3)*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:

```
sage: timeit('fc_wilk_zz(30)') # random, long time
625 loops, best of 3: 15.4 us per loop
```

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:
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.)

We also have support for RR:

And support for CDF:

Currently, fast_callable() can accept two kinds of objects: polynomials (univariate and multivariate) and symbolic expressions (elements of the Symbolic Ring). (This list is likely to grow significantly in the near future.) For polynomials, you can omit the ‘vars’ argument; the variables will default to the ring generators (in the order used when creating the ring).

But you can also specify the variable names to override the default ordering (you can include extra variable names here, too).

For symbolic expressions, you need to specify the variable names, so that fast_callable() knows what order to use.
You can also specify extra variable names:

```python
sage: ff = fast_callable(f, vars=('x', 'w', 'z', 'y'))
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 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('z')
```

Expression trees support Python’s numeric operators, so you can easily build expression trees representing arithmetic expressions.

```python
sage: v1 = (x+y) * (y+z) + (y//z)
```

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
sin(add(add(mul(add(add(v_0, v_1), add(v_1, v_1)), floordiv(v_1, v_1)), add(mul(3.1415900000000000, v_0), mul(1729, v_1)))),
    add(mul(add(v_0, v_1), add(v_1, v_1)), floordiv(v_1, 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(add(v_0, v_1), add(v_1, v_1)), floordiv(v_1, v_1)), add(mul(3.1415900000000000, v_0), mul(1729, v_1))))
sage: fast_callable(e)(1, 2, 3)
3.60555127546399
```

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:

```
sage: var('x')
x
sage: f = fast_callable(sqrt(x^7+1), vars=[x], domain=float)
sage: f(1)
1.4142135623730951
sage: f.op_list()
```

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:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
def etb = ExpressionTreeBuilder(‘x’)

sage: x = etb.var(‘x’)
sage: f = etb.call(sqrt, x^7 + 1)
sage: g = etb.call(sin, x)
sage: fast_callable(f+g).op_list()
```

AUTHOR:

• Carl Witty (2009-02): initial version (heavily inspired by Robert Bradshaw’s fast_eval.pyx)

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 `COND ? T : F` expression or the Python `if COND else F`. This lets you define piecewise functions using `fast_callable()`.

```
sage: v4 = etb.choice(v3 >= etb.constant(0), v1, v2)  # not tested
```

The arguments are `(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.

```
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)
```

Now `ff` 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))
```

but this code will only evaluate x+1 once:

```python
sage: v = x+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.
```

```python
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).
```

```python
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)))
```
abs()

Compute the absolute value of an Expression.

EXAMPLES:

```python
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)
```

class sage.ext.fast_callable.ExpressionCall

Bases: `sage.ext.fast_callable.Expression`

An Expression that represents a function call.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: etb.call(sin, x).arguments()
[v_0]
```

class sage.ext.fast_callable.ExpressionChoice

Bases: `sage.ext.fast_callable.Expression`

A conditional expression.

(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 = ExpressionTreeBuilder(vars=(x,))
sage: etb.call(sin, x).function()
sin
```
sage: etb.choice(etb.call(operator.eq, x, 0), 0, 1/x)
(0 if {eq}(v_0, 0) else div(1, v_0))

condition()
Return the condition of an ExpressionChoice.
EXAMPLES:

sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: x = etb(x)
sage: etb.choice(x, ~x, 0).condition()
v_0

if_false()
Return the false branch of an ExpressionChoice.
EXAMPLES:

sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: x = etb(x)
sage: etb.choice(x, ~x, 0).if_false()
0

if_true()
Return the true branch of an ExpressionChoice.
EXAMPLES:

sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: x = etb(x)
sage: etb.choice(x, ~x, 0).if_true()
inv(v_0)

class sage.ext.fast_callable.ExpressionConstant
Bases: sage.ext.fast_callable.Expression
An Expression that represents an arbitrary constant.
EXAMPLES:

sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: type(etb(3))
<type 'sage.ext.fast_callable.ExpressionConstant'>

value()
Return the constant value of an ExpressionConstant.
EXAMPLES:
class sage.ext.fast_callable.ExpressionIPow
Bases: sage.ext.fast_callable.Expression

A power Expression with an integer exponent.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: type(etb.var('x')^17)
<type 'sage.ext.fast_callable.ExpressionIPow'>
```

def base()
    Return the base from this ExpressionIPow.

    EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
33
```

def exponent()
    Return the exponent from this ExpressionIPow.

    EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
-1
```

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)
```

def 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, notices if the function is operator.pow and the second argument is integral, and constructs
    an ExpressionIPow instead.

2.10. Fast Expression Evaluation
EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: etb.call(cos, x)
cos(v_0)
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)
```

-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 = ExpressionTreeBuilder(vars=(x,))
sage: etb.choice(etb.call(operator.eq, x, 0), 0, 1/x)
(0 if {eq}(v_0, 0) else div(1, v_0))
```

-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)
pi
sage: etb = ExpressionTreeBuilder('x', domain=RealField(200))
sage: etb.constant(pi)
3.1415926535897932384626433832795028841971693993751058209749
```

-var(`v``)

Turn the argument into an ExpressionVariable. Look it up in the list of variables. (Variables are matched by name.)

EXAMPLES:

```python
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))
sage: etb.var(some_really_long_name)
v_2
sage: etb.var('some_really_long_name')
v_2
```

(continues on next page)
sage: etb.var(x)
 v_3
sage: etb.var('y')
Traceback (most recent call last):
 ... 
ValueError: Variable 'y' not found

class sage.ext.fast_callable.ExpressionVariable
Bases: sage.ext.fast_callable.Expression

An Expression that represents a variable.

EXAMPLES:

sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: type(etb.var(x))
<type 'sage.ext.fast_callable.ExpressionVariable'>

variable_index()
Return the variable index of an ExpressionVariable.

EXAMPLES:

sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: etb(x).variable_index()
0

class sage.ext.fast_callable/InstructionStream
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:

sage: from sage.ext.interpreters.wrapper_rdf import metadata
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('py_call', math.sin, 1)
sage: instr_stream.instr('abs')
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_rdf import metadata
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}
```

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:

```python
sage: from sage.ext.interpreters.wrapper_rdf import metadata
sage: from sage.ext.fast_callable import InstructionStream
sage: instr_stream = InstructionStream(metadata, 1)
sage: md = instr_stream.get_metadata()
sage: type(md)
<type 'sage.ext.fast_callable.InterpreterMetadata'>
```

has_instr(opname)
Check whether this InstructionStream knows how to generate code for a given instruction.

EXAMPLES:
```python
sage: from sage.ext.interpreters.wrapper_rdf import metadata
sage: from sage.ext.fast_callable import InstructionStream
sage: instr_stream = InstructionStream(metadata, 1)
True
sage: instr_stream.has_instr('factorial')
False
sage: 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:**

```python
sage: from sage.ext.interpreters.wrapper_rdf import metadata
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')
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()
[('load_arg', 0), 'sin', ('py_call', <built-in function sin>, 1), 'abs', 'return']
```

**`load_arg(n)`**

Add a ‘load_arg’ instruction to this InstructionStream.

**EXAMPLES:**

```python
sage: from sage.ext.interpreters.wrapper_rdf import metadata
sage: from sage.ext.fast_callable import InstructionStream
sage: instr_stream = InstructionStream(metadata, 12)
sage: instr_stream.load_arg(5)
sage: instr_stream.current_op_list()
[('load_arg', 5)]
sage: instr_stream.load_arg(3)
sage: instr_stream.current_op_list()
[('load_arg', 5), ('load_arg', 3)]
```

**`load_const(c)`**

Add a ‘load_const’ instruction to this InstructionStream.

**EXAMPLES:**

```python
sage: from sage.ext.interpreters.wrapper_rdf import metadata
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)]

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]
```

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)
pi^2

sage: square(I)
-1

sage: square(RIF(-1, 1)).str(style='brackets')
'[-0.0000000000000000 .. 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
```

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<=n<=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:

```sage
sage: fast_callable(sin(x)/x, vars=[x], domain=RDF).get_orig_args()
{'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:

```sage
sage: fast_callable(cos(x)*x, vars=[x], domain=RDF).op_list()
[(('load_arg', 0), ('load_arg', 0), 'cos', 'mul', 'return')]
```

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:

```sage
sage: fast_callable(abs(sin(x)), vars=[x], domain=RDF).python_calls()
[]
sage: fast_callable(abs(sin(factorial(x))), vars=[x]).python_calls()
[factorial, sin]
```

`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: 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:

```
sage: f_float = fast_callable(expr, vars=[x], domain=float)
sage: f_float(2)
12.909297426825681
sage: f_rdf = fast_callable(expr, vars=[x], domain=RDF)
sage: f_rdf(2)
12.909297426825681
sage: f_cdf = fast_callable(expr, vars=[x], domain=CDF)
sage: f_cdf(2)
12.909297426825681
sage: f_cdf(2+I)
10.40311925062204 + 11.510943740958707*I
```

```
sage: f = fast_callable(expr, vars=('z','x','y'))
sage: f(1, 2, 3)
sin(2) + 12
```

```
sage: K.<x> = QQ[]
sage: p = K.random_element(6); p
-1/4*x^6 + 1/2*x^5 - x^4 - 12*x^3 + 1/2*x^2 - 1/95*x - 1/2
sage: fp = fast_callable(p, domain=RDF)
sage: fp.op_list()
[(load_arg, 0), (load_const, -0.25), 'mul', ('load_const', 0.5), 'add', ('load_arg', 0), 'mul', ('load_const', -1.0), 'add', ('load_arg', 0), 'mul', ('load_const', -12.0), 'add', ('load_const', 0), 'mul', ('load_const', -0.5), 'add', 'return']
sage: fp(3.14159)
-552.4182988917153
```

```
sage: f = fast_callable(p, domain=RDF)
sage: f.op_list()
# py2
[(load_const, 0.0), ('load_const', -1.0), ('load_const', 0.0), ('ipow', 1), ('load_arg', 2), ('ipow', 1), 'mul', 'mul', 'add', ('load_const', 1.0), ('load_arg', 0), ('ipow', 1), ('load_arg', 1), ('load_arg', 2), 'mul', 'mul', 'add', ('load_const', 0.0), 'add', ('load_const', 1), ('ipow', 2), 'mul', 'add', ('load_const', -1.0), ('load_arg', 1), ('ipow', 1), ('load_arg', 2), ('ipow', 1), 'mul', 'mul', 'add', 'return']
```

(continues on next page)
sage: fp.op_list()  # py3
[['load_const', 0.0], ['load_const', 1.0], ['load_arg', 0], ['ipow', 1],
 'load_arg', 1, ['ipow', 2], 'mul', 'mul', 'add', ['load_const', 0.3333333333333333],
 ('load_arg', 1), ('ipow', 2), 'mul', 'add', ['load_const', -1.0], ['load_arg', 0],
 ('ipow', 1), ('load_arg', 2), ('ipow', 1), 'mul', 'mul', 'add', ['load_const', -1.0],
 ('load_arg', 1), ('ipow', 1), ('load_arg', 2), ('ipow', 1), 'mul', 'mul', 'add',
 'return']
sage: fp(e, pi, sqrt(2))  # abs tol 3e-14
21.831120464939584
sage: symbolic_result = p(e, pi, sqrt(2)); symbolic_result
pi^2*e + 1/3*pi^2 - sqrt(2)*pi - sqrt(2)*e
sage: n(symbolic_result)
21.8311204649396
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=('x', 'y'), domain=float)
sage: x = etb.var('x')
sage: y = etb.var('y')
sage: expr = etb.call(sin, x^2 + y); expr
sin(add(ipow(v_0, 2), v_1))
sage: fc = fast_callable(expr, domain=float)
sage: fc(5, 7)
0.5514266812416906
Check that fast_callable also works for symbolic functions with evaluation functions:

sage: def evalf_func(self, x, y, parent):
    return parent(x*y) if parent is not None,
    else x*y
sage: f = function('f', evalf_func=evalf_func)
sage: fc = fast_callable(f(x, y), vars=[x, y])
f(3, 4)
And also when there are complex values involved:

sage: def evalf_func(self, x, y, parent):
    return parent(I*x*y) if parent is not None,
    else I*x*y
sage: g = function('g', evalf_func=evalf_func)
sage: fc = fast_callable(g(x, y), vars=[x, y])
g(3, 4)
g(3, 4)
sage: fc2 = fast_callable(g(x, y), domain=complex, vars=[x, y])
sage: fc2(3, 4)
12j
sage: fc3 = fast_callable(g(x, y), domain=float, vars=[x, y])
sage: fc3(3, 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.all.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:

```python
sage: from sage.ext.fast_callable import function_name
sage: function_name(operator.pow)
'pow'
sage: function_name(cos)
'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 ExpressionTreeBuilder 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
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)
<type 'sage.ext.interpreters.wrapper_py.Wrapper_py'>
sage: v(7)
8*pi + 56
```

**sage.ext.fast_callable.get_builtin_functions()**

To handle ExpressionCall, we need to map from Sage and Python functions to opcode names.

This returns a dictionary which is that map.

We delay building builtin_functions to break a circular import between sage.calculus and this file.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import get_builtin_functions
sage: builtins = get_builtin_functions()
sage: sorted(set(builtins.values()))
(continues on next page)
```
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_rdf import metadata
sage: from sage.ext.fast_callable import InstructionStream, op_list
sage: instr_stream = InstructionStream(metadata, 1)
sage: instr_stream.instr('load_arg', 0)
sage: instr_stream.instr('abs')
sage: instr_stream.instr('return')
sage: instr_stream.current_op_list()
[['load_arg', 0], 'abs', 'return']
sage: op_list(instr_stream.get_current(), metadata)
[['load_arg', 0], 'abs', 'return']
```

### 2.10.2 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.

Up until now the solution has been to use lambda expressions, but this is neither intuitive, Sage-like, nor efficient (compared to operating on raw C doubles). This module provides a representation of algebraic expression in Reverse Polish Notation, and provides an efficient interpreter on C double values as a callable python object. It does what it can in C, and will call out to Python if necessary.

Essential to the understanding of this class is the distinction between symbolic expressions and callable symbolic expressions (where the latter binds argument names to argument positions). The *vars parameter passed around encapsulates this information.

See the function `fast_float(f, *vars)` to create a fast-callable version of f.

**Note:** Sage temporarily has two implementations of this functionality; one in this file, which will probably be deprecated soon, and one in fast_callable.pyx. The following instructions are for the old implementation; you probably want to be looking at fast_callable.pyx instead.
To provide this interface for a class, implement `fast_float_(self, *vars)`. The basic building blocks are provided by the functions `fast_float_constant` (returns a constant function), `fast_float_arg` (selects the n-th value when called with \( \geq n \) arguments), and `fast_float_func` which wraps a callable Python function. These may be combined with the standard Python arithmetic operators, and support many of the basic math functions such `sqrt`, `exp`, and trig functions.

EXAMPLES:

```python
sage: from sage.ext.fast_eval import fast_float
sage: f = fast_float(sqrt(x^7+1), 'x', old=True)
sage: f(1)
1.4142135623730951
sage: f.op_list()
['load 0', 'push 7.0', 'pow', 'push 1.0', 'add', 'call sqrt(1)']
```

To interpret that last line, we load argument 0 (x in this case) onto the stack, push the constant 2.0 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_eval import fast_float_arg
sage: g = fast_float_arg(0).sin()
sage: (f+g).op_list()
['load 0', 'push 7.0', 'pow', 'push 1.0', 'add', 'call sqrt(1)', 'load 0', 'call sin(1)', 'add']
```

AUTHORS:

• Robert Bradshaw (2008-10): Initial version

```python
class sage.ext.fast_eval.FastDoubleFunc
Bases: object

This class is for fast evaluation of algebraic expressions over the real numbers (e.g. for plotting). It represents an expression as a stack-based series of operations.

EXAMPLES:

```python
sage: from sage.ext.fast_eval import FastDoubleFunc
sage: f = FastDoubleFunc('const', 1.5) # the constant function
sage: f()
1.5
sage: g = FastDoubleFunc('arg', 0) # the first argument
sage: g()
0
sage: h = f+g
sage: h(17)
18.5
sage: h = h.sin()
```

We can wrap Python functions too:
```python
sage: list(h)
['push 1.5', 'load 0', 'add', 'call sin(1)']
```
```python
sage: h = FastDoubleFunc('callable', lambda x,y: x*x*x - y, g, f)
```

```python
sage: h(10)
998.5
```

```python
sage: h.is_pure_c()
False
```

```python
sage: list(h)
['load 0', 'push 1.5', 'py_call <function <lambda> at 0x...>(2)'
```

Here's a more complicated expression:

```python
sage: from sage.ext.fast_eval import fast_float_constant, fast_float_arg
sage: a = fast_float_constant(1.5)
sage: b = fast_float_constant(3.14)
sage: c = fast_float_constant(7)
sage: x = fast_float_arg(0)
sage: y = fast_float_arg(1)
sage: f = a*x^2 + b*x + c - y/sqrt(sin(y)^2+a)
sage: f(2,3)
16.846610528508116
```

```python
sage: f.max_height
4
```

```python
sage: f.is_pure_c()
True
```

```python
sage: list(f)
['push 1.5', 'load 0', 'dup', 'mul', 'mul', 'push 3.14', 'load 0', 'mul', 'add',
 → 'push 7.0', 'add', 'load 1', 'load 1', 'call sin(1)', 'dup', 'mul', 'push 1.5',
 → 'add', 'call sqrt(1)', 'div', 'sub']
```

AUTHORS:

- Robert Bradshaw

**abs()**

EXAMPLES:

```python
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).abs()
sage: f(3)
3.0
```

**arccos()**

EXAMPLES:

```python
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).arccos()
sage: f(sqrt(3)/2)
0.5235987755982989...
```

**arccosh()**

EXAMPLES:

```python
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).arccosh()
sage: f(cosh(5))
5.0
```
arcsin()
EXAMPLES:

```python
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).arcsin()
sage: f(0.5)
0.523598775598298...
```

arcsinh()
EXAMPLES:

```python
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).arcsinh()
sage: f(sinh(5))
5.0
```

arctan()
EXAMPLES:

```python
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).arctan()
sage: f(1)
0.785398163397448...
```

arctanh()
EXAMPLES:

```python
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).arctanh()
sage: abs(f(tanh(0.5)) - 0.5) < 0.0000001
True
```

ceil()
EXAMPLES:

```python
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).ceil()
sage: f(1.5)
2.0
sage: f(-1.5)
-1.0
```

cos()
EXAMPLES:

```python
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).cos()
sage: f(0)
1.0
```

cosh()
EXAMPLES:

```python
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).cosh()
```
(continues on next page)
sage: f(log(2))
1.25

cot()
EXAMPLES:

```
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).cot()
sage: f(pi/4)
1.0...
```

csc()
EXAMPLES:

```
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).csc()
sage: f(pi/2)
1.0
```

exp()
EXAMPLES:

```
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).exp()
sage: f(1)
2.718281828459045...
sage: f(100)
2.6881171418161356e+43
```

floor()
EXAMPLES:

```
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).floor()
sage: f(11.5)
11.0
sage: f(-11.5)
-12.0
```

is_pure_c()
Returns True if this function can be evaluated without any python calls (at any level).

EXAMPLES:

```
sage: from sage.ext.fast_eval import fast_float_constant, fast_float_arg, fast_float_func
sage: fast_float_constant(2).is_pure_c()
True
sage: fast_float_arg(2).sqrt().sin().is_pure_c()
True
sage: fast_float_func(lambda x: 2).is_pure_c()
False
```
log(base=None)
EXAMPLES:

```
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).log()
sage: f(2)
0.693147180559945...
sage: f = fast_float_arg(0).log(2)
sage: f(2)
1.0
sage: f = fast_float_arg(0).log(3)
sage: f(9)
2.0...
```

max_height

nargs

nops

op_list()
Returns a list of string representations of the operations that make up this expression.
Python and C function calls may be only available by function pointer addresses.
EXAMPLES:

```
sage: from sage.ext.fast_eval import fast_float_constant, fast_float_arg
sage: a = fast_float_constant(17)
sage: x = fast_float_arg(0)
sage: a.op_list()
['push 17.0']
sage: x.op_list()
['load 0']
sage: (a*x).op_list()
['push 17.0', 'load 0', 'mul']
sage: (a+a*x**2).sqrt().op_list()
['push 17.0', 'push 17.0', 'load 0', 'dup', 'mul', 'mul', 'add', 'call sqrt(1)']
```

python_calls()
Returns a list of all python calls used by function.
EXAMPLES:

```
sage: from sage.ext.fast_eval import fast_float_func, fast_float_arg
sage: x = fast_float_arg(0)
sage: f = fast_float_func(hash, sqrt(x))
sage: f.op_list()
['load 0', 'call sqrt(1)', 'py_call <built-in function hash>(1)']
sage: f.python_calls()
[<built-in function hash>]
```

sec()
EXAMPLES:

```
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).sec()
```

(continues on next page)
sage: f(pi)
-1.0

**sin()**

**EXAMPLES:**

```
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).sin()
sage: f(pi/2)
1.0
```

**sinh()**

**EXAMPLES:**

```
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).sinh()
sage: f(log(2))
0.75
```

**sqrt()**

**EXAMPLES:**

```
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).sqrt()
sage: f(4)
2.0
```

**tan()**

**EXAMPLES:**

```
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).tan()
sage: f(pi/3)
1.73205080756887...
```

**tanh()**

**EXAMPLES:**

```
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0).tanh()
sage: f(0)
0.0
```

---

**sage.ext.fast_eval.fast_float**

Tries to create a function that evaluates f quickly using floating-point numbers, if possible. There are two implementations of fast_float in Sage; by default we use the newer, which is slightly faster on most tests.

On failure, returns the input unchanged.

**INPUT:**

- f – an expression
- vars – the names of the arguments
- old – use the original algorithm for fast_float
• expect_one_var – don’t give deprecation warning if vars is omitted, as long as expression has only one var

EXAMPLES:

```
sage: from sage.ext.fast_eval import fast_float
sage: x,y = var('x,y')
sage: f = fast_float(sqrt(x^2+y^2), 'x', 'y')
sage: f(3,4)
5.0
```

Specifying the argument names is essential, as fast_float objects only distinguish between arguments by order.

```
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
```

```
sage.ext.fast_eval.fast_float_arg(n)
```

Return a fast-to-evaluate argument selector.

INPUT:

• n – the (zero-indexed) argument to select

EXAMPLES:

```
sage: from sage.ext.fast_eval import fast_float_arg
sage: f = fast_float_arg(0)
sage: f(1,2)
1.0
sage: f = fast_float_arg(1)
sage: f(1,2)
2.0
```

This is all that goes on under the hood:

```
sage: fast_float_arg(10).op_list()
['load 10']
```

```
sage.ext.fast_eval.fast_float_constant(x)
```

Return a fast-to-evaluate constant function.

EXAMPLES:

```
sage: from sage.ext.fast_eval import fast_float_constant
sage: f = fast_float_constant(-2.75)
sage: f()
-2.75
```

This is all that goes on under the hood:

```
sage: fast_float_constant(pi).op_list()
['push 3.1415926535...']
```
sage.ext.fast_eval.fast_float_func(f, *args)
Returns a wrapper around a python function.

INPUT:

• f – a callable python object

• args – a list of FastDoubleFunc inputs

EXAMPLES:

```python
sage: from sage.ext.fast_eval import fast_float_func, fast_float_arg
sage: f = fast_float_arg(0)
sage: g = fast_float_arg(1)
sage: h = fast_float_func(lambda x,y: x-y, f, g)
sage: h(5, 10)
-5.0
```

This is all that goes on under the hood:

```python
sage: h.op_list()
['load 0', 'load 1', 'py_call <function <lambda> at 0x...>(2)']
```

sage.ext.fast_eval.is_fast_float(x)
CHAPTER
THREE

CODE EVALUATION

3.1 Cython support functions

AUTHORS:

• William Stein (2006-01-18): initial version
• William Stein (2007-07-28): update from sagex to cython
• Martin Albrecht & William Stein (2011-08): cfile & cargs

`sage.misc.cython.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:

```python
sage: from sage.misc.cython import compile_and_load
sage: module = compile_and_load("def f(int n):
               return n^n")
sage: module.f(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

`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 trac ticket #12446.

**EXAMPLES:**

We create a Lambda function in pure Python (using the `r` to make sure the 3.2 is viewed as a Python float):

```python
sage: f = lambda x,y: x*x + y*y + x + y + 17*r*x + 3.2r
```

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
g(2,3)
55.2
sage: g(0,0)
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
sage: a = 25
sage: f(10)
24.45597888911063
sage: a = 50
sage: f(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:**

```python
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'
```
3.2 Fortran compiler

```
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:
    sage: code = '''
    ....: C FILE: FIB1.F
    ....: SUBROUTINE FIB(A,N)
    ....: 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)
    ....:     ENDF
    ....:  ENDDO
    ....: END
    ....: C END FILE FIB1.F'''
    sage: fortran(code, globals())
    sage: import numpy
    sage: a = numpy.array(range(10), dtype=float)
    sage: fib(a, 10)
    sage: 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)

**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:

```python
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
```

**set_names(new_names)**

**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()

sage: p.parse("1+2")
3
sage: p.parse("1+2 == 3")
True

sage: p = Parser(make_var=var)
```

(continues on next page)
sage: p.parse("a*b^c - 3a")
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, make_function={
    'foo': (lambda x: x*x+x)})
sage: p.parse("1.5 + foo(b)")
b^2 + b + 1.50000000000000
sage: p.parse("1.9").parent()
Real Field with 53 bits of precision

\texttt{p\_arg()}\texttt{(tokens)}

Returns an expr, or a (name, expr) tuple corresponding to a single function call argument.

\textbf{EXAMPLES:}

Parsing a normal expression:

\begin{verbatim}
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)
sage: p.p_arg(Tokenizer("a+b"))
a + b
\end{verbatim}

A keyword expression argument:

\begin{verbatim}
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)
sage: p.p_arg(Tokenizer("val=a+b"))
('val', a + b)
\end{verbatim}

A lone list::

\begin{verbatim}
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)
sage: p.p_arg(Tokenizer("[x]"))
[x]
\end{verbatim}

\texttt{p\_args()}\texttt{(tokens)}

Returns a list, dict pair.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser()
sage: p.p_args(Tokenizer("1,2,a=3"))
[[1, 2], {'a': 3}]
\end{verbatim}
**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()"))
((), {})
```

**p_eqn(tokens)**

Parse an equation or expression.

This is the top-level node called by the `parse` function.

**EXAMPLES:**

```python
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)
sage: p.p_eqn(Tokenizer("1+a"))
a + 1
sage: p.p_eqn(Tokenizer("a == b"))
a == b
sage: p.p_eqn(Tokenizer("a < b"))
a < b
sage: p.p_eqn(Tokenizer("a > b"))
a > b
sage: p.p_eqn(Tokenizer("a <= b"))
a <= b
sage: p.p_eqn(Tokenizer("a >= b"))
a >= b
sage: p.p_eqn(Tokenizer("a != b"))
a != b
```

**p_expr(tokens)**

Parse a list of one or more terms.

3.3. A parser for symbolic equations and expressions 235
EXAMPLES:

```python
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)
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"))
-a^2 + a - b + 4*c
sage: p.p_expr(Tokenizer("a - -3"))
a + 3
sage: 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:

```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("- -t"))
t
sage: p.p_factor(Tokenizer("- + - -t^2"))
-t^2
sage: p.p_factor(Tokenizer("t^11 * x"))
t^11
```

**p_list(tokens)**

Parse a list of items.

EXAMPLES:

```python
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)

sage: p.p_list(Tokenizer(['[1+2, 1e3]']))
[3, 1000.0]
sage: p.p_list(Tokenizer('[]'))
[]
```

**p_matrix(tokens)**

Parse a matrix.

EXAMPLES:

```python
sage: from sage.misc.parser import Parser, Tokenizer
sage: p = Parser(make_var=var)

sage: p.p_matrix(Tokenizer(['[a,0],[0,a]']))
[a 0]
[0 a]
```

**p_power(tokens)**

Parses a power. Note that exponentiation groups right to left.

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("-(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("t**2"))
t^2
sage: p.p_power(Tokenizer("2^3^2")) == 2^9
True
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

p_sequence(tokens)
Parse a (possibly nested) set of lists and tuples.

EXAMPLES:

sage: from sage.misc.parser import Parser, Tokenizer
def p_sequence(tokens):
    return p_sequence(Parser(make_var=var))(Tokenizer(tokens))
sage: p_sequence(Tokenizer("[[1+2,0],(1,2,3),[1+a,2+b,(3+c),(4+d,)]"]"))
[[[3, 0], (1, 2, 3), [a + 1, b + 2, c + 3, (d + 4,)]]]

p_term(tokens)
Parse a single term (consisting of one or more factors).

EXAMPLES:

sage: from sage.misc.parser import Parser, Tokenizer
def p_term(tokens):
    return p_term(Parser(make_var=var))(Tokenizer(tokens))
sage: p_term(Tokenizer("a*b"))
a*b
sage: p_term(Tokenizer("a * b / c * d"))
a*b*c*d/c
sage: p_term(Tokenizer("-a * b + c"))
-a*b
sage: p_term(Tokenizer("a*(b-c)^2")
a*(b - c)^2
sage: p_term(Tokenizer("-3*a"))
-3*a

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)
```

**parse**(*s*, *accept_eqn=True*)

Parse the given string.

EXAMPLES:

```python
sage: from sage.misc.parser import Parser
sage: p = Parser(make_var=var)
(sage: p.parse("E = m c^2"))
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)
(sage: p.parse_expression('a-3b^2'))
-3*b^2 + a
```

**parse_sequence**(*s*)

Parse a (possibly nested) set of lists and tuples.

EXAMPLES:

```python
sage: from sage.misc.parser import Parser
sage: p = Parser(make_var=var)
(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
```

**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
sage: from sage.misc.parser import Tokenizer
sage: Tokenizer("1.5+2*3^4-sin(x)").test()
['FLOAT(1.5)', '+', 'INT(2)', '*', 'INT(3)', '^', 'INT(4)', '-', 'NAME(sin)', '(', 'NAME(x)', ')']
```

The single character tokens are given by:

```python
sage: Tokenizer("+-*/^(),=<\["]".test()
['+', '-', '*', '/', '^', '(', ')', '<', '>', '[', ']', '{', '}']
```
Two-character comparisons accepted are:

```
sage: Tokenizer("<= >= != == **").test()
['LESS_EQ', 'GREATER_EQ', 'NOT_EQ', '==', '^']
```

Integers are strings of 0-9:

```
sage: Tokenizer("1 123 9879834759873452908375013").test()
['INT(1)', 'INT(123)', 'INT(9879834759873452908375013)'
```

Floating point numbers can contain a single decimal point and possibly exponential notation:

```
sage: 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:

```
sage: Tokenizer("-1 -1.2").test()
['-', 'INT(1)', '-', 'FLOAT(1.2)'
```

Names are alphanumeric sequences not starting with a digit:

```
sage: Tokenizer("a a1 _a_24").test()
['NAME(a)', 'NAME(a1)', 'NAME(_a_24)'
```

There is special handling for matrices:

```
sage: Tokenizer("matrix(a)").test()
['MATRIX', '(' , 'NAME(a)', ' )'
```

Anything else is an error:

```
sage: Tokenizer("&@~").test()
['ERROR', 'ERROR', 'ERROR']
```

No attempt for correctness is made at this stage:

```
sage: Tokenizer("\( 5e5\)").test()
['(' , 'FLOAT(5e5)' , ')'
```

```
sage: Tokenizer("?%%").test()
['ERROR', 'ERROR', 'ERROR']
```

```
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:
sage: from sage.misc.parser import Tokenizer, token_to_str
sage: t = Tokenizer("a+b")
sage: token_to_str(t.next())
'NAME'
sage: token_to_str(t.next())
'+'
sage: t.backtrack()  # the return type is bint for performance reasons
```

(continues on next page)
False

```python
sage: token_to_str(t.next())
'+'
```

### last()

Returns the last token seen.

**EXAMPLES:**

```python
sage: from sage.misc.parser import Tokenizer, token_to_str
sage: t = Tokenizer("3a")
sage: token_to_str(t.next())
'INT'
sage: token_to_str(t.last())
'INT'
sage: token_to_str(t.next())
'NAME'
sage: 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")
sage: token_to_str(t.next())
'NAME'
sage: t.last_token_string()
'a'
sage: token_to_str(t.next())
'-'
sage: token_to_str(t.next())
'FLOAT'
sage: 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")
sage: token_to_str(t.next())
'NAME'
sage: token_to_str(t.next())
'+'
sage: token_to_str(t.next())
'INT'
sage: token_to_str(t.next())
'EOS'
```
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")
sage: token_to_str(t.peek())
'NAME'
sage: token_to_str(t.next())
'NAME'
sage: token_to_str(t.peek())
'+'
sage: token_to_str(t.peek())
'+'
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")
sage: t.test()  # Output: ['NAME(a)', '+', 'NAME(b)', '*', 'NAME(c)']
sage: t.test()  # Output: []
sage: t.reset()
sage: t.test()  # Output: ['NAME(a)', '+', 'NAME(b)', '*', 'NAME(c)']
sage: t.reset(3)
sage: t.test()  # Output: ['*', '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()  # Output: ['INT(12345)', '+', 'NAME(a)']
sage: t.reset(2)
sage: t.test()  # Output: ['INT(345)', '+', 'NAME(a)']
```

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()  # Output: ['NAME(a)', 'NAME(b)', 'INT(3)']
```
sage: t.test()  
[]

`sage.misc.parser.foo(*args, **kwds)`

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})
```

`sage.misc.parser.token_to_str(token)`

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 preparsering

`class sage.misc.python.Python`  
Bases: `object`

Allows for evaluating a chunk of code without any preparsering.

`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:**

```python
sage: from sage.misc.python import Python
sage: python = Python()
sage: python.eval('2+2', globals())
4
```

Any variables that are set during evaluation of the block will propagate to the `globals` dictionary.
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():\n    return 'foo'\nend mumble():\n    print('mumble {}'.format(foo()))\nmumble", globals())
foo
mumble foo
```

```
sage: 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.
This example illustrates setting a variable for use in evaluation.

```python
sage: x = 5
sage: eval('4/3 + x', {'x': 25})  # py2
26
sage: eval('4//3 + x', {'x': 25})  # py3
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', cmgs='K.<x> = QQ[]\np = x^2 + 1'))
x^2 + 1
```

If you give commands to execute and a dictionary of variables, then the dictionary will be modified by assignments in the commands:

```python
sage: vars = {}
sage: sage_eval('None', cmgs='y = 3', locals=vars)
sage: vars['y'], parent(vars['y'])
(3, Integer Ring)
```

You can also specify the object to evaluate as a tuple. A 2-tuple is assumed to be a pair of a command sequence and an expression; a 3-tuple is assumed to be a triple of a command sequence, an expression, and a dictionary holding local variables. (In this case, the given dictionary will not be modified by assignments in the commands.)

```python
sage: sage_eval(('f(x) = x^2', 'f(3)'))
9
sage: vars = {'rt2': sqrt(2.0)}
sage: sage_eval(('rt2 += 1', 'rt2', vars))
2.41421356237309
sage: vars['rt2']
1.41421356237310
```

This example illustrates how `sage_eval` can be useful when evaluating the output of other computer algebra systems.
Here you can see eval simply will not work but `sage_eval` will.

```python
sage.misc.sage_eval.sageobj(x, vars=None)
```

Return a native Sage object associated to `x`, if possible and implemented.

If the object has an `_sage_` method it is called and the value is returned. Otherwise `str` is called on the object, and all preparsing is applied and the resulting expression is evaluated in the context of `from sage.all import *`.

To evaluate the expression with certain variables set, use the `vars` argument, which should be a dictionary.

**EXAMPLES:**

```python
sage: type(sageobj(gp('34/56')))
<type 'sage.rings.rational.Rational'>
sage: n = 5/2
sage: sageobj(n) is n
True
sage: k = sageobj('Z(8^3/1)', {'Z':ZZ}); k
512
sage: type(k)
<type 'sage.rings.integer.Integer'>
```

This illustrates interfaces:

```python
sage: f = gp('2/3')
sage: type(f)
<class 'sage.interfaces.gp.GpElement'>
sage: f._sage_()
2/3
sage: type(f._sage_())
<type 'sage.rings.rational.Rational'>
sage: a = gap(939393/2433)
sage: a._sage_()
313131/811
sage: type(a._sage_())
<type 'sage.rings.rational.Rational'>
```
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:

```python
sage: sh.eval('''echo "Hello there"\nif [ $? -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(u'')
sage: for i in range(1, 5):
    ....:    symbols += ascii_left_parenthesis.character_art(i)
    ....:    symbols += ascii_art(u' ')
    ....:    symbols += ascii_right_parenthesis.character_art(i)
    ....:    symbols += ascii_art(u' ')
sage: for i in range(1, 5):
    ....:    symbols += ascii_left_square_bracket.character_art(i)
    ....:    symbols += ascii_art(u' ')
    ....:    symbols += ascii_right_square_bracket.character_art(i)
    ....:    symbols += ascii_art(u' ')
sage: for i in range(1, 5):
    ....:    symbols += ascii_left_curly_brace.character_art(i)
    ....:    symbols += ascii_art(u' ')
    ....:    symbols += ascii_right_curly_brace.character_art(i)
    ....:    symbols += ascii_art(u' ')
sage: symbols
( ) [ ] { }
( ) ( ) [ ] [ ] [ ] { } { } { }
( ) ( ) ( ) [ ] [ ] [ ] [ ] [ ] { } { } { } { }
```

```python
sage: symbols = unicode_art(u'')
sage: for i in range(1, 5):
    ....:    symbols += unicode_left_parenthesis.character_art(i)
    ....:    symbols += unicode_art(u' ')
    ....:    symbols += unicode_right_parenthesis.character_art(i)
    ....:    symbols += unicode_art(u' ')
sage: for i in range(1, 5):
    ....:    symbols += unicode_left_square_bracket.character_art(i)
    ....:    symbols += unicode_art(u' ')
    ....:    symbols += unicode_right_square_bracket.character_art(i)
    ....:    symbols += unicode_art(u' ')
sage: symbols
(continues on next page)
```

(continues on next page)
symbols += unicode_art(u' ')
sage: for i in range(1, 5):
    symbols += unicode_left_curly_brace.character_art(i)
    symbols += unicode_right_curly_brace.character_art(i)
    symbols += unicode_art(u' ')

sage:
\[
\begin{array}{c}
\{ } \{ } \{ } \\
( ) ( ) ( ) \\
[ ] [ ] [ ] \\
\end{array}
\]

class sage.typeset.symbols.CompoundAsciiSymbol(character, top, extension, bottom, middle=None, middle_top=None, middle_bottom=None, top_2=None, bottom_2=None):
    Bases: sage.typeset.symbols.CompoundSymbol

def character_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)
{ }
{ }

class sage.typeset.symbols.CompoundSymbol(character, top, extension, bottom, middle=None, middle_top=None, middle_bottom=None, top_2=None, bottom_2=None):
    Bases: sage.structure.sage_object.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:**

```python
sage: from sage.typeset.symbols import CompoundSymbol
sage: i = CompoundSymbol('I', '+', '|', '+', '|')
sage: i.print_to_stdout(1)
I
sage: i.print_to_stdout(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)
∫
sage: unicode_integral.print_to_stdout(2)
∫
∫
sage: unicode_integral.print_to_stdout(3)
∫
∫
∫
sage: unicode_integral.print_to_stdout(4)
∫
∫
∫
```

**class** `sage.typeset.symbols.CompoundUnicodeSymbol`

Bases: `sage.typeset.symbols.CompoundSymbol`

**character_art(num_lines)**

Return the unicode art of the symbol

**EXAMPLES:**

```python
sage: from sage.typeset.symbols import *
sage: unicode_left_curly_brace.character_art(3)
{ } { }
```
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.

```python
class sage.typeset.character_art.CharacterArt(lines=[], breakpoints=[], baseline=None):
    Bases: sage.structure.sage_object.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')
sage: ascii_art(sum(pi^i/factorial(i)*x^i, i, 0, oo))
    pi*x
e
```

```python
classmethod empty()
    Return the empty character art object

    EXAMPLES:

    sage: from sage.typeset.ascii_art import AsciiArt
    sage: AsciiArt.empty()
```

```python
get_baseline()
    Return the line where the baseline is, for example:

    5 4
    14*x + 5*x

    the baseline has at line 0 and

    { 0 }
    { \ : 4 }
    { 0 }

    has at line 1.
```

```python
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 trac ticket #29204.

    For example the expression:
```
The expression $14x + 5x$ can be split on position 4 (on the `+`).

**EXAMPLES:**

```python
sage: from sage.typeset.ascii_art import AsciiArt
sage: p3 = AsciiArt([" * ", "***"])
sage: p5 = AsciiArt([" * ", " * * ", "*****"])
sage: aa = ascii_art([p3, p5])
sage: aa.get_breakpoints()
doctest:...: DeprecationWarning: get_breakpoints() is deprecated
See https://trac.sagemath.org/29204 for details.
[6]
```

**height()**

Return the height of the ASCII art object.

**OUTPUT:**

Integer. The number of lines.

**split(pos)**

Split the representation at the position `pos`.

**EXAMPLES:**

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

**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

class sage.typeset.character_art_factory.CharacterArtFactory(art_type, string_type, magic_method_name, parenthesis, square_bracket, curly_brace)

Bases: sage.structure.sage_object.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
def 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:

```python
sage: ascii_art(integral(exp(x+x^2)/(x+1), x))
```

build_container(content, left_border, right_border, baseline=0)

Return character art for a container.

INPUT:
• content – \texttt{CharacterArt}; the content of the container, usually comma-separated entries
• left\_border – \texttt{CompoundSymbol}; the left border of the container
• right\_border – \texttt{CompoundSymbol}; the right border of the container
• baseline – (default: 0) the baseline of the object

\textbf{build\_dict}(d, baseline=0)

Return a character art output of a dictionary.

\textbf{build\_empty}()

Return the empty character art object

\textbf{OUTPUT:}

Character art instance.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.typeset.ascii_art import \_ascii_art_factory as factory
sage: str(factory.build_empty())
'
\end{verbatim}

\begin{verbatim}
\textbf{build\_from\_magic\_method}(obj, baseline=None)

Return the character art object created by the object’s magic method

\textbf{OUTPUT:}

Character art instance.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.typeset.ascii_art import \_ascii_art_factory as factory
sage: out = factory.build_from_magic_method(identity_matrix(2)); out
[1 0]
[0 1]
sage: type(out)
<class 'sage.typeset.ascii_art.AsciiArt'>
\end{verbatim}

\begin{verbatim}
\textbf{build\_from\_string}(obj, baseline=0)

Return the character art object created from splitting the object’s string representation.

\textbf{INPUT:}

• obj – utf-8 encoded byte string or unicode
• baseline – (default: 0) the baseline of the object

\textbf{OUTPUT:}

Character art instance.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.typeset.ascii_art import \_ascii_art_factory as factory
sage: out = factory.build_from_string('a\nbb\ncccc')
sage: out + out + out
a a a
bb bb bb
cccccc
sage: type(out)
<class 'sage.typeset.ascii_art.AsciiArt'>
\end{verbatim}

4.1. Formatted Output
build_list($l$, $baseline=0$)
Return a character art output of a list.

build_set($s$, $baseline=0$)
Return a character art output of a set.

build_tuple($t$, $baseline=0$)
Return a character art output of a tuple.

concatenate($iterable$, $separator$, $empty=None$, $baseline=0$, $nested=False$)
Concatenate multiple character art instances
The breakpoints are set as the breakpoints of the separator together with the breakpoints of the objects in iterable. If there is None, the end of the separator is used.

INPUT:
- iterable – iterable of character art
- separable – character art; the separator in-between the iterable
- empty – an optional character art which is returned if iterable is empty
- baseline – (default: 0) the baseline of the object
- nested – boolean (default: 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:

```sage
i2 = identity_matrix(2)
sage: ascii_art(i2, i2, i2, sep=ascii_art(1/x))
1 1
[1 0]-[1 0]-[1 0]
[0 1]|x[0 1]|x[0 1]
```

parse_keywords($kwds$)
Parse the keyword input given by the dict $kwds$.

INPUT:
- $kwds$ – 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/separater and baseline from kwds if they are specified.
4.1.4 ASCII Art

This file contains:

- `AsciiArt` an simple implementation of an ASCII art object,
- `ascii_art()` a function to get the ASCII art representation of any object in Sage,

**AUTHOR:**

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

**EXAMPLES:**

```python
sage: n = var('n')
sage: integrate(n^2/x,x)
n^2*log(x)
sage: ascii_art(integrate(n^2/x,x))
2
n*log(x)
sage: ascii_art(integrate(n^2/(pi*x),x))
\frac{2n \cdot log(x)}{\pi}

sage: ascii_art(list(Partitions(6)))
[ * ]
[ ** * ]
[ *** ** * * ]
[ **** **** * *** ** * * * * ]
[ ***** **** * *** ** * ** * * * * ]
```

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

```
%display ascii_art
```

```python
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('%display ascii_art')
sage: i = var('i')
sage: shell.run_cell('sum(factorial(i)*x^i, i, 0, 10)')
10 9 8 7 6 5 4 3
3628800*x + 362880*x + 40320*x + 5040*x + 720*x + 120*x + 24*x + 6*x + 2*x + x + 1
sage: shell.run_cell('3/(7*x)')
3
\frac{3}{7x}
```

```python
sage: shell.run_cell('list(Compositions(5))')
[ * ]
[ * ** * ]
[ * * ** *** * ** * * ]
[ * * * * ** ** *** **** * * ** *** * ** * * ]
[ * * * * * * * * * * * * * * * * * * * * * * * ]
```

(continues on next page)
```python
sage: shell.run_cell('%display simple')
sage: shell.quit()
```

(continues on next page)
class sage.typeset.ascii_art.AsciArt(lines=[], breakpoints=[], baseline=None)

Bases: sage.typeset.character_art.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
class sage.typeset.ascii_art.AsciArt:
    def __init__(self, lines=None, breakpoints=None, baseline=None):
        self.lines = lines or []
        self.breakpoints = breakpoints or []
        self.baseline = baseline or 0

    def ascii_art(*obj, sep=' ', baseline=0):
        return ascii_art(*obj, sep=sep, baseline=baseline)
```

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

`sage.typeset.ascii_art.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:

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

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=' : ')
[1 0 0]
[1 0]
[1 0 0]
[0 1 0]
[0 0 1]
```

We can specify the baseline:

```python
sage: ascii_art(ident(2), baseline=-1) + ascii_art(ident(3))
[1 0][1 0 0]
[0 1][0 1 0]
[0 0 1]
```

We can determine the baseline of the separator:

```python
sage: ascii_art(ident(1), ident(2), ident(3), sep=' -- ', sep_baseline=-1)
[1 0 0]
[1 0]
[1 0 0]
[0 1 0]
[0 0 1]
```

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)
```

```
```

258 Chapter 4. Formatted Output
4.1.5 Unicode Art

This module implements ascii art using unicode characters. It is a strict superset of `ascii_art`.

```
class sage.typeset.unicode_art.UnicodeArt(lines=[], breakpoints=[], baseline=None)
```

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')
sage: unicode_art(sum(pi^i/factorial(i)*x^i, i, 0, oo))
pix xe
```

```
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:**

```python
sage: unicode_art(integral(exp(sqrt(x))/(x+pi), x))
∫x√xe dx
sage: ident = lambda n: identity_matrix(ZZ, n)
sage: unicode_art(ident(1), ident(2), ident(3), sep=': ')
 1 0 0
 0 1 0
 0 0 1
```

If specified, the `sep_baseline` overrides the baseline of an unicode art separator:

4.1. Formatted Output
sage.typeset.unicode_art.unicode_subscript(x)
Return the integer \( x \) as a superscript.

EXAMPLES:

```python
sage: from sage.typeset.unicode_art import unicode_subscript
sage: unicode_subscript(15123902)
'15123902'
sage: unicode_subscript(-712)
'−712'
```

sage.typeset.unicode_art.unicode_superscript(x)
Return the rational number \( x \) as a superscript.

EXAMPLES:

```python
sage: from sage.typeset.unicode_art import unicode_superscript
sage: unicode_superscript(15123902)
'15123902'
sage: unicode_superscript(-712/5)
'−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
sage: from sage.typeset.unicode_characters import unicode_otimes
sage: unicode_otimes
'⊗'
sage: hex(ord(_))
'0x2297'
sage: from sage.typeset.unicode_characters import unicode_bigotimes
sage: unicode_bigotimes
'⨂'
sage: hex(ord(_))
'0x2a02'
sage: from sage.typeset.unicode_characters import unicode_wedge
sage: unicode_wedge
'\wedge'
```

(continues on next page)
sage: unicode_wedge
'∧'
sage: hex(ord(_))
'0x2227'
sage: from sage.typeset.unicode_characters import unicode_bigwedge
sage: unicode_bigwedge
'\∧'
sage: hex(ord(_))
'0x22c0'
sage: from sage.typeset.unicode_characters import unicode_partial
sage: unicode_partial
'\partial'
sage: hex(ord(_))
'0x2202'

Arrows:

sage: from sage.typeset.unicode_characters import unicode_to
sage: unicode_to
'\to'
sage: hex(ord(_))
'0x2192'
sage: from sage.typeset.unicode_characters import unicode_mapsto
sage: unicode_mapsto
'\mapsto\to'
sage: hex(ord(_))
'0x21a6'

Letters:

sage: from sage.typeset.unicode_characters import unicode_mathbbR
sage: unicode_mathbbR
'\bb R'
sage: hex(ord(_))
'0x211d'
sage: from sage.typeset.unicode_characters import unicode_mathbbC
sage: unicode_mathbbC
'\bb C'
sage: hex(ord(_))
'0x2102'

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)
'(-x^2)'
sage: coeff_repr(QQ(1/2), is_latex=True)
'\frac{1}{2}'
sage: coeff_repr(-x^2, is_latex=True)
'\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: '' if scalar_mult is '*'; otherwise scalar_mult)
- `coeffs` – for backward compatibility

**OUTPUT:**
- `str` - a string

**EXAMPLES:**

```python
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`:
4.1.8 Sage Input Formatting

This module provides the function \texttt{sage_input()} that takes an arbitrary sage value and produces a sequence of commands that, if typed at the \texttt{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 \texttt{sage_input()} to find a simple way to produce the same value. We attempt to produce commands that are readable and idiomatic:

\begin{verbatim}
    sage: sage_input(3)
    3
    sage: sage_input((polygen(RR) + RR(pi))^2, verify=True)
    # Verified
    R.<x> = RR[]
x^2 + 6.2831853071795862*x + 9.869604401089358
\end{verbatim}
With verify=True, sage_input() also verifies the results, by calling sage_eval() on the result and verifying that it is equal to the input.:

```
sage: sage_input(GF(2)(1), verify=True)
# Verified
GF(2)(1)
```

We can generate code that works without the preparser, with preparse=False; or we can generate code that will work whether or not the preparser is enabled, with preparse=None. Generating code with preparse=False may be useful to see how to create a certain value in a Python or Cython source file.:

```
sage: sage_input(5, verify=True)
# Verified
5
sage: sage_input(5, preparse=False)
ZZ(5)
sage: sage_input(5, preparse=None)
ZZ(5)
sage: sage_input(5r, verify=True)
# Verified
5r
sage: sage_input(5r, preparse=False)
5
sage: sage_input(5r, preparse=None)
int(5)
```

Adding sage_input() support to your own classes is straightforward. You need to add a _sage_input_() method which returns a SageInputExpression (henceforth abbreviated as SIE) which will reconstruct this instance of your class.

A _sage_input_ method takes two parameters, conventionally named sib and coerced. The first argument is a SageInputBuilder; it has methods to build SIEs. The second argument, coerced, is a boolean. This is only useful if your class is a subclass of Element (although it is always present). If coerced is False, then your method must generate an expression which will evaluate to a value of the correct type with the correct parent. If coerced is True, then your method may generate an expression of a type that has a canonical coercion to your type; and if coerced is 2, then your method may generate an expression of a type that has a conversion to your type.

Let's work through some examples. We'll build a sequence of functions that would be acceptable as _sage_input_ methods for the Rational class.

Here's the first and simplest version.:

```
sage: def qq_sage_input_v1(self, sib, coerced):
    ....:     return sib(self.numerator())/sib(self.denominator())
```

We see that given a SageInputBuilder sib, you can construct a SIE for a value v simply with sib(v), and you can construct a SIE for a quotient with the division operator. Of course, the other operators also work, and so do function calls, method calls, subscripts, etc.

We'll test with the following code, which you don't need to understand. (It produces a list of 8 results, showing the formatted versions of -5/7 and 3, with the preparser either enabled or disabled and either with or without an automatic coercion to QQ.):

```
sage: from sage.misc.sage_input import SageInputBuilder
sage: def test_qq_formatter(fmt):
    ....:     results = []
```
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, \(-\text{ZZ}(5)/\text{ZZ}(7) == -\text{ZZ}(5)/\text{ZZ}(7)\):

```python
sage: def qq_sage_input_v2(self, sib, coerced):
    ....:    return sib(self.numerator())/sib.int(self.denominator())
```

The `int` method on SageInputBuilder returns a SIE for an integer that is always represented in the simple way, without coercions. (So, depending on the preparser mode, it might read in as an `Integer`, an `int`, or a `long`):

```python
sage: test_qq_formatter(qq_sage_input_v2)
[-\text{ZZ}(5)/7, -\text{ZZ}(5)/7, -5/7, -5/7, \text{ZZ}(3)/1, \text{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 sage constant or function:

```python
sage: test_qq_formatter(qq_sage_input_v3)
[-\text{ZZ}(5)/7, -\text{ZZ}(5)/7, -5/7, -5/7, \text{QQ}(3), 3, \text{QQ}(3), 3]
```

This is the prettiest output we're going to get, but let's make one further refinement. Other `_sage_input_` methods, like the one for polynomials, analyze the structure of SIEs; they work better (give prettier output) if negations are at the outside. If the above code were used for rationals, then `sage_input(polygen(QQ) - 2/3)` would produce \(x + (-2/3)\); if we change to the following code, then we would get \(x - 2/3\) instead:

```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(self.numerator())/sib.int(self.denominator())
```

(continues on next page)
....:     v = sib(num)/sib.int(self.denominator())
....:     if neg: v = -v
....:     return v

sage: test_qq_formatter(qq_sage_input_v4)
[-ZZ(5)/7, -ZZ(5)/7, -5/7, -5/7, QQ(3), 3, QQ(3), 3]

AUTHORS:

- Carl Witty (2008-04): new file
- Vincent Delecroix (2015-02): documentation formatting

class sage.misc.sage_input.SIE_assign(sib, lhs, rhs)
    Bases: sage.misc.sage_input.SageInputExpression
    This class represents an assignment command.
    EXAMPLES:

    sage: from sage.misc.sage_input import SageInputBuilder
    sage: sib = SageInputBuilder()
    sage: sib.assign(sib.name('foo').x, sib.name('pi'))
    {assign: {getattr: {atomic:foo}.x} {atomic:pi}}

class sage.misc.sage_input.SIE_binary(sib, op, lhs, rhs)
    Bases: sage.misc.sage_input.SageInputExpression
    This class represents an arithmetic expression with a binary operator and its two arguments, in a sage_input() expression tree.
    EXAMPLES:

    sage: from sage.misc.sage_input import SageInputBuilder
    sage: sib = SageInputBuilder()
    sage: sib(3)+5
    {binop:+ {atomic:3} {atomic:5}}

class sage.misc.sage_input.SIE_call(sib, func, args, kwargs)
    Bases: sage.misc.sage_input.SageInputExpression
    This class represents a function-call node in a sage_input() expression tree.
    EXAMPLES:

    sage: from sage.misc.sage_input import SageInputBuilder
    sage: sib = SageInputBuilder()
    sage: sie = sib.name('GF')
    sage: sie(49)
    {call: {atomic:GF}{{atomic:49}}}
EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: sib.dict([(('TeX', RR(pi)), ('Metafont', RR(e))]
{dict: {{atomic:'TeX'}:{call: {atomic:RR}({atomic:3.1415926535897931})}, {atomic:
˓→'Metafont'}:{call: {atomic:RR}({atomic:2.7182818284590451})}}}
```

```python
sage: sib.dict({-40:-40, 0:32, 100:212})
{dict: {{unop:- {atomic:40}}:{unop:- {atomic:40}}, {atomic:0}:{atomic:32},
˓→{atomic:100}:{atomic:212}}}
```

```python
class sage.misc.sage_input.SIE_gen(sib, parent, name)
Bases: sage.misc.sage_input.SageInputExpression

This class represents a named generator of a parent with named generators.

EXAMPLES:

```python
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',)}}
```

```python
class sage.misc.sage_input.SIE_gens_constructor(sib, constr, gen_names, gens_syntax=None)
Bases: sage.misc.sage_input.SageInputExpression

This class represents an expression that can create a sage parent with named generators, optionally using the sage preparser generators syntax (like K.<x> = QQ[]).

EXAMPLES:

```python
sage: from sage.misc.sage_input import 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',)}
```

```python
class sage.misc.sage_input.SIE_getattr(sib, obj, attr)
Bases: sage.misc.sage_input.SageInputExpression

This class represents a getattr node in a `sage_input()` expression tree.

EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sie = sib.name('CC').gen()
sage: sie
{call: {getattr: {atomic:CC}.gen}()}
```

```python
class sage.misc.sage_input.SIE_import_name(sib, module, name, alt_name=None)
Bases: sage.misc.sage_input.SageInputExpression
```
This class represents a name which has been imported from a module.

EXAMPLES:

```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}
```

class sage.misc.sage_input.SIE_literal(sib)

Bases: sage.misc.sage_input.SageInputExpression

An abstract base class for literals (basically, values which consist of a single token).

EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder, SIE_literal
sage: sib = SageInputBuilder()
sage: isinstance(sib(3), SIE_literal)
True
sage: isinstance(sib(3.14159, True), SIE_literal)
True
sage: isinstance(sib.name('pi'), SIE_literal)
True
sage: isinstance(sib(False), SIE_literal)
True
sage: sib(False)
{atomic:False}
```

class sage.misc.sage_input.SIE_literal_stringrep(sib, n)

Bases: sage.misc.sage_input.SIE_literal

Values in this class are leaves in a `sage_input()` expression tree. Typically they represent a single token, and consist of the string representation of that token. They are used for integer, floating-point, and string literals, and for name expressions.

EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder, SIE_literal_stringrep
sage: sib = SageInputBuilder()
sage: isinstance(sib(3), SIE_literal_stringrep)
True
sage: isinstance(sib(3.14159, True), SIE_literal_stringrep)
True
sage: isinstance(sib.name('pi'), SIE_literal_stringrep)
True
sage: isinstance(sib(False), SIE_literal_stringrep)
True
sage: sib(False)
{atomic:False}
```

class sage.misc.sage_input.SIE_subscript(sib, coll, key)

Bases: sage.misc.sage_input.SageInputExpression

This class represents a subscript node in a `sage_input()` expression tree.

EXAMPLES:
```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: sie = sib.name('QQ')['x,y']
sage: sie
{subscr: {atomic:QQ}[{atomic:'x,y']}}
```

**class sage.misc.sage_input.SIE_tuple(sib, values, is_list)**

Bases: `sage.misc.sage_input.SageInputExpression`

This class represents a tuple or list node in a `sage_input()` expression tree.

**EXAMPLES:**

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: sib((1, 'howdy'))
{tuple: ({atomic:1}, {atomic:'howdy'})}
sage: sib(['lists'])
{list: ({atomic:'lists'})}
```

**class sage.misc.sage_input.SIE_unary(sib, op, operand)**

Bases: `sage.misc.sage_input.SageInputExpression`

This class represents an arithmetic expression with a unary operator and its argument, in a `sage_input()` expression tree.

**EXAMPLES:**

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: -sib(256)
{unop:- {atomic:256}}
```

**class sage.misc.sage_input.SageInputAnswer**

Bases: tuple

This class inherits from tuple, so it acts like a tuple when passed to `sage_eval()`; but it prints as a sequence of commands.

**EXAMPLES:**

```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\n'
'sage: v[1]
'x/7'
sage: len(v)
2
```

(continues on next page)
sage: v = SageInputAnswer('', 'sin(3.14)', {'sin': math.sin}); v

LOCALS:
  sin: <built-in function sin>
sin(3.14)
sage: v[0]
''
sage: v[1]
'sin(3.14)'
sage: v[2]
{'sin': <built-in function sin>}

class sage.misc.sage_input.SageInputBuilder(allow_locals=False, preparse=True)

Bases: object

An instance of this class is passed to _sage_input_ methods. It keeps track of the current state of the _sage_input_ process, and contains many utility methods for building SageInputExpression objects.

In normal use, instances of SageInputBuilder are created internally by sage_input(), but it may be useful to create an instance directly for testing or doctesting.

EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder
```

We can create a SageInputBuilder, use it to create some SageInputExpression s, and get a result. (As mentioned above, this is only useful for testing or doctesting; normally you would just use sage_input()):

```python
sage: sib = SageInputBuilder()
sage: sib.result((sib(3) + sib(4)) * (sib(5) + sib(6)))
(3 + 4)*(5 + 6)
```

**assign**(e, val)

Constructs a command that performs the assignment e=val.

Can only be used as an argument to the command method.

**INPUT:**

- e, val – SageInputExpression

**EXAMPLES:**

```python
sage: from sage.misc.sage_input import SageInputBuilder
```

```python
sage: sib = SageInputBuilder()
sage: si = [None]
sage: si[0] = si
```

**cache**(x, sie, name)

**INPUT:**

- x - an arbitrary value
- sie - a SageInputExpression
• name - a requested variable name

Enters \( x \) and \( \text{sie} \) in a cache, so that subsequent calls \( \text{self}(x) \) will directly return \( \text{sie} \). Also, marks the requested name of this \( \text{sie} \) to be name.

This should almost always be called as part of the method \( \text{self._sage_input_} \) 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()

sage: sie42 = sib(GF(101)(42))

sage: sib.cache(GF(101)(42), sie42, 'the_ultimate_answer')

sage: sib.result(sib(GF(101)(42)) + sib(GF(101)(42)))

the_ultimate_answer = GF(101)(42)
the_ultimate_answer + the_ultimate_answer
```

Note that we don’t assign the result to a variable if the value is only used once.

```python
sage: sib = SageInputBuilder()

sage: sie42 = sib(GF(101)(42))

sage: sib.cache(GF(101)(42), sie42, 'the_ultimate_answer')

sage: sib.result(sib(GF(101)(42)) + sib(GF(101)(43)))

GF_101 = GF(101)
GF_101(42) + GF_101(43)
```

**command**(\( v, \text{cmd} \))

INPUT:

• \( v, \text{cmd} \) – SageInputExpression

Attaches a command to \( v \), which will be executed before \( v \) is used. Multiple commands will be executed in the order added.

EXAMPLES:

```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]))

sage: sib.result(incr_list)

si = []
si.append(1)
si.extend([2, 3])
si
```

dict(\( \text{entries} \))

Given a dictionary, or a list of (key, value) pairs, produces a \textit{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}
sage: sib.result(sib.dict([('hello', 'sunshine'), ('goodbye', 'rain')]))
{'hello': 'sunshine', 'goodbye': 'rain'}
```

empty_subscript(parent)

Given a `SageInputExpression` representing `foo`, produces a `SageInputExpression` representing `foo[]`. Since this is not legal Python syntax, it is useful only for producing the sage generator syntax for a polynomial ring.

EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: sib.result(sib.empty_subscript(sib(2) + sib(3)))
(2 + 3)[]
```

The following calls this method indirectly:

```python
sage: sage_input(polygen(ZZ['y']))
R.<x> = ZZ['y'][]
x
```

float_str(n)

Given a string representing a floating-point number, produces a `SageInputExpression` that formats as that string.

EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: sib.result(sib.float_str(repr(RR(e))))
2.71828182845905
```

gen(parent, n=0)

Given a parent, returns a `SageInputExpression` for the `n`-th (default 0) generator of the parent.

EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: sib.result(sib.gen(ZZ['y']))
R.<y> = ZZ[
```

ggetattr(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).
id_cache(x, sie, name)

INPUT:

• x - an arbitrary value  
• sie - a SageInputExpression  
• name - a requested variable name

Enters x and sie in a cache, so that subsequent calls self(x) will directly return sie. Also, marks the requested name of this sie to be name. Differs from the method{cache} method in that the cache is keyed by id(x) instead of by x.

This may be called on values of an arbitrary type, which may be useful if the values are both large and likely to be used multiple times in a single expression; it should be preferred to method{cache} if equality on the values is difficult or impossible to compute.

EXAMPLES:

```sage
def import_name(module, name, alt_name=None):
    # Implementation
    pass
```

4.1. Formatted Output
• module, name, alt_name – strings

Creates an expression that will import a name from a module and then use that name.

EXAMPLES:

```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
```

`int(n)`

Return a raw SIE from the integer n.

As it is raw, it may read back as a Sage Integer, a Python int or a Python long, depending on its size and whether the preparser is enabled.

INPUT:

• n - a Sage Integer, a Python int or a Python long

EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()

sage: sib.result(sib.int(-3^50))
-717897987691852588770249

sage: sib = SageInputBuilder()

sage: sib.result(sib.int(-42r))
-42
```

`name(n)`

Given a string representing a Python name, produces a `SageInputExpression` for that name.

EXAMPLES:

```python
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 \( K.<x> = \mathbb{Q}[x] \)).

The method \(_{\text{sage input}}\) method of a parent class with generators should construct a \emph{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
....:     name = 'the_thing'
....:     result = sib.parent_with_gens(parent, normal_sie,
....:                                    gen_names, name,
....:                                    gens_syntax=gens_sie)
....:     return sib, result

sage: sib, par_sie = test_setup()
sage: sib.result(par_sie)
make_a_parent(names=('foo', 'bar'))

sage: sib, par_sie = test_setup()
sage: sib.result(sib(3) * sib.gen("some parent", 0))
the_thing.<foo,bar> = make_a_parent()
3*foo

sage: sib, par_sie = test_setup(preparse=False)
sage: sib.result(par_sie)
make_a_parent(names=('foo', 'bar'))

sage: sib, par_sie = test_setup(preparse=False)
sage: sib.result(sib(3) * sib.gen("some parent", 0))
the_thing = make_a_parent(names=('foo', 'bar'))
foo,bar = the_thing.gens()
ZZ(3)*foo

sage: sib, par_sie = test_setup(use_gens=False)
sage: sib.result(par_sie)
make_a_parent(names=('foo', 'bar'))

sage: sib, par_sie = test_setup(use_gens=False)
```
sage: sib.result(sib(3) * sib.gen("some parent", 0))
the_thing = make_a_parent(names=('foo', 'bar'))
foo, bar = the_thing.gens()
3*foo

sage: sib, par_sie = test_setup()
sage: sib.result(par_sie - sib.gen("some parent", 1))
the_thing.<foo,bar> = make_a_parent()
the_thing - bar

**prepare()**

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:**

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: SageInputBuilder().preparse()
True
sage: SageInputBuilder(preparse=False).preparse()
False
```

**prod(factors, simplify=False)**

Given a sequence, returns a *SageInputExpression* for the product of the elements.

With simplify=True, performs some simplifications first. If any element is formatted as a string '0', then that element is returned directly. If any element is formatted as a string '1', then it is removed from the sequence (unless it is the only element in the sequence). And any negations are removed from the elements and moved to the outside of the product.

**EXAMPLES:**

```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
```

(continues on next page)
sage: sib = SageInputBuilder()
sage: sib.result(sib.prod([1, 1, 1], simplify=True))
1

result(e)
Given a SageInputExpression constructed using self, returns a tuple of a list of commands and an
expression (and possibly a dictionary of local variables) suitable for sage_eval().

EXAMPLES:

sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: r = sib.result(sib(6) * sib(7)); r
6*7
sage: tuple(r)
('', '6*7')

share(sie)
Mark the given expression as sharable, so that it will be replaced by a variable if it occurs multiple times
in the expression. (Most non-single-token expressions are already sharable.)

EXAMPLES:

sage: from sage.misc.sage_input import SageInputBuilder
Without explicitly using .share(), string literals are not shared:

sage: sib = SageInputBuilder()
sage: e = sib('hello')
sage: sib.result(sib((e, e)))
('hello', 'hello')

See the difference if we use .share:

sage: sib = SageInputBuilder()
sage: e = sib('hello')
sage: sib.share(e)
sage: sib.result(sib((e, e)))
('hello', 'hello')

si = 'hello'
(si, si)

sum(terms, simplify=False)
Given a sequence, returns a SageInputExpression for the product of the elements.

With simplify=True, performs some simplifications first. If any element is formatted as a string '0',
then it is removed from the sequence (unless it is the only element in the sequence); and any instances of a
+ -b are changed to a - b.

EXAMPLES:

sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: sib.result(sib.sum([-1, 0, 1, 0, -1]))

-1 + 0 + 1 + 0 -1

sage: sib = SageInputBuilder()
sage: sib.result(sib.sum([-1, 0, 1, 0, -1], simplify=True))
-1 + 1 - 1

sage: sib = SageInputBuilder()
sage: sib.result(sib.sum([0, 0, 0], simplify=True))
0

use_variable(sie, name)

Marks the SageInputExpressionsie 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()
```

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()
```

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)
```

class sage.misc.sage_input.SageInputExpression(sib)

Bases: object

Subclasses of this class represent expressions for sage_input(). sage classes should define a method{_sage_input_} method, which will return an instance of SageInputExpression, created using methods of SageInputBuilder.

To the extent possible, operations on SageInputExpression objects construct a new SageInputExpression representing that operation. That is, if a is a SageInputExpression, then a + b constructs a SageInputExpression representing this sum. This also works for attribute access, function calls, subscripts, etc. Since arbitrary attribute accesses might be used to construct a new attribute-access expression, all internal attributes and methods have names that begin with _sie_ to reduce the chance of collisions.

It is expected that instances of this class will not be directly created outside this module; instead, instances will be created using methods of SageInputBuilder and SageInputExpression.
Values of type `SageInputExpression` print in a fairly ugly way, that reveals the internal structure of the expression tree.

```python
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 `sage_input()` formatting process.

```
format(e, prec)
Format a Sage input expression into a string.

INPUT:

• e - a `SageInputExpression`
• prec - an integer representing a precedence level

First, we check to see if e should be replaced by a variable. If so, we generate the command to assign the variable, and return the name of the variable.

Otherwise, we format the expression by calling its method `_sie_format` method, and add parentheses if necessary.

EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder, SageInputFormatter
sage: sib = SageInputBuilder()
sage: sif = SageInputFormatter()
sage: sie = sib(GF(5))

Here we cheat by calling method `_sie_prepare` twice, to make it use a variable:

```python
sage: sie._sie_prepare(sif)
sage: sie._sie_prepare(sif)
sage: sif._commands ''
sage: sif.format(sie, 0)
'GF_5'
sage: sif._commands
'GF_5 = GF(5)\n'

We demonstrate the use of commands, by showing how to construct code that will produce a random matrix:

```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 ''
sage: sif.format(sie, 0)
'si'
sage: sif._commands
'si = matrix(ZZ, 10, 10)\nsi.randomize()\n'
```
get_name(name)
Return a name corresponding to a given requested name. If only one request for a name is received, then we will use the requested name; otherwise, we will add numbers to the end of the name to make it unique.

If the input name is None, then it is treated as a name of 'si'.

EXAMPLES:

```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'
```

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
(set(), {'x': 0})

sage: sif.register_name('x')

sage: sif._names, sif._dup_names
(set(), {'x': 0})
```

sage.misc.sage_input.sage_input(x, preparse=True, verify=False, allow_locals=False)
Return a sequence of commands that can be used to rebuild the object x.

INPUT:

- x - the value we want to find an input form for
- preparse - (default True) Whether to generate code that requires the preparser. With True, generated code requires the preparser. With False, generated code requires that the preparser not be used. With None, generated code will work whether or not the preparser is used.
- verify - (default False) If True, then the answer will be evaluated with sage_eval(), and an exception will be raised if the result is not equal to the original value. (In fact, for verify=True, sage_input() is effectively run three times, with preparse set to True, False, and None, and all three results are checked.) This is particularly useful for doctests.
- allow_locals - (default False) If True, then values that sage_input() cannot handle are returned in a dictionary, and the returned code assumes that this dictionary is passed as the locals parameter of sage_eval(). (Otherwise, if sage_input() cannot handle a value, an exception is raised.)
### EXAMPLES:

```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))
```

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
```

The result of `sage_input()` is actually a pair of strings with a special `__repr__` method to print nicely:.

```python
sage: r = sage_input(RealField(20)(pi), verify=True)
sage: r
# Verified
RealField(20)(3.1415939)
sage: isinstance(r, tuple)
True
sage: len(r)
2
sage: tuple(r)
('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: Can't convert <function <lambda> at 0x...> to sage_input form
```

But we can have `sage_input()` continue anyway, and return an input form for the rest of the expression, with `allow_locals=True`:

```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)
('RealField(20)(3.1415939)', '{'_sil1': <function <lambda> at 0x...>}')
```

`sage.misc.sage_input.verify_same(a, b)`

Verify that two Sage values are the same. This is an extended equality test; it checks that the values are equal.
and that their parents are equal. (For values which are not Elements, the types are checked instead.)

If the values are the same, we return None; otherwise, we raise an exception.

EXAMPLES:

```
sage: from sage.misc.sage_input import verify_same
sage: verify_same(1, 1)
Traceback (most recent call last):
AssertionError: Expected 1 == 2
sage: verify_same(1, 1r)
Traceback (most recent call last):
AttributeError: 'int' object has no attribute 'parent'
sage: verify_same(1r, 1)
Traceback (most recent call last):
  assert(type(a) == type(b))
AssertionError
sage: verify_same(5, GF(7)(5))
Traceback (most recent call last):
  assert(a.parent() == b.parent())
AssertionError
```

```
sage.misc.sage_input.verify_si_answer(x, answer, preparse)
```

Verify that evaluating `answer` gives a value equal to `x` (with the same parent/type). If `preparse` is True or False, then we evaluate `answer` with the preparser enabled or disabled, respectively; if `preparse` is None, then we evaluate `answer` both with the preparser enabled and disabled and check both results.

On success, we return None; on failure, we raise an exception.

INPUT:

- `x` - an arbitrary Sage value
- `answer` - a string, or a `SageInputAnswer`
- `preparse` - True, False, or None

EXAMPLES:

```
sage: from sage.misc.sage_input import verify_si_answer
sage: verify_si_answer(1, '1', True)
sage: verify_si_answer(1, '1', False)
Traceback (most recent call last):
  ... AttributeError: 'int' object has no attribute 'parent'
sage: verify_si_answer(1, 'ZZ(1)', None)
```

4.1.9 Tables

Display a rectangular array as a table, either in plain text, LaTeX, or html. See the documentation for \texttt{table} for details and examples.

AUTHORS:
  - John H. Palmieri (2012-11)

\textbf{class} \texttt{sage.misc.table.table}(\texttt{rows=} None, \texttt{columns=} None, \texttt{header_row=} False, \texttt{header_column=} False, \texttt{frame=} False, \texttt{align=} 'left')

Bases: \texttt{sage.structure.sage_object.SageObject}

Display a rectangular array as a table, either in plain text, LaTeX, or html.

INPUT:

- \texttt{rows} (default None) - a list of lists (or list of tuples, etc.), containing the data to be displayed.
- \texttt{columns} (default None) - a list of lists (etc.), containing the data to be displayed, but stored as columns. Set either \texttt{rows} or \texttt{columns}, but not both.
- \texttt{header_row} (default False) - if True, first row is highlighted.
- \texttt{header_column} (default False) - if True, first column is highlighted.
- \texttt{frame} (default False) - if True, put a box around each cell.
- \texttt{align} (default 'left') - the alignment of each entry: either 'left', 'center', or 'right'

EXAMPLES:

\begin{verbatim}
\textit{sage:} rows = [['a', 'b', 'c'], [100,2,3], [4,5,60]]
\textit{sage:} table(rows)
   a  b  c
  100 2 3
   4  5 60
\textit{sage:} latex(table(rows))
\begin{tabular}{lll}
a & b & c \\
$100$ & $2$ & $3$ \\
$4$ & $5$ & $60$ \\
\end{tabular}
\end{verbatim}

If \texttt{header_row} is True, then the first row is highlighted. If \texttt{header_column} is True, then the first column is highlighted. If \texttt{frame} is True, then print a box around every “cell”.

\begin{verbatim}
\textit{sage:} table(rows, header_row=True)
   a  b  c
+--------+-+-----+
  100 2 3
   4  5 60
\textit{sage:} latex(table(rows, header_row=True))
\begin{tabular}{lll}
a & b & c \\
\hline
$100$ & $2$ & $3$ \\
$4$ & $5$ & $60$ \\
\end{tabular}
\textit{sage:} table(rows=rows, frame=True)
\end{verbatim}

(continues on next page)
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.

You can create the transpose of this table in several ways, for example, “by hand,” that is, changing the data defining the table:
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)
```

```
+-----------++------+------+------+------+
| $x$ || 0 | 1 | 2 | 3 |
+-----------++------+------+------+------+
| $\sin(x)$ || 0.00 | 0.84 | 0.91 | 0.14 |
+-----------++------+------+------+------+
```

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()
```

```
+-----+---+----+
| a   | b  | c   |
| 100  | 2  | 3   |
| 4  | 5 | 60 |
+-----+---+----+
```

In either plain text or LaTeX, entries in tables can be aligned to the left (default), center, or right:

```python
data = [['$x$', r'$\sin(x)$'] + [(x, n(sin(x), digits=2)) for x in [0..3]] for i in [0..3]]
```

```python
sage: table(rows=data, header_row=True, frame=True)
```

```
+-----------++------+------+------+------+
| $x$ || 0 | 1 | 2 | 3 |
+-----------++------+------+------+------+
| $\sin(x)$ || 0.00 | 0.84 | 0.91 | 0.14 |
+-----------++------+------+------+------+
```

```
+-----+---+----+
| a   | b  | c   |
| 100  | 2  | 3   |
| 4  | 5 | 60 |
+-----+---+----+
```

To generate HTML you should use `html(table(...))`:

```python
data = [['$x$', r'$\sin(x)$'] + [(x, n(sin(x), digits=2)) for x in [0..3]] for i in [0..3]]
sage: output = html(table(data, header_row=True, frame=True))
sage: type(output)
<class 'sage.misc.html.HtmlFragment'>
```
It is an error to specify both rows and columns:

```
sage: table(rows=[[1,2,3], [4,5,6]], columns=[[0,0,0], [0,0,1024]])
Traceback (most recent call last):
...
ValueError: Don't set both 'rows' and 'columns' when defining a table.
```

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
```
_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
sage: dm = 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:

```
sage: T = table([[‘a’, ‘b’, ‘c’], [1,2,3]])
sage: T.options(align=’right’, frame=True)
sage: T.options(header_row=False)
```

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.

```
sage: T = table([[1,2,3], [4,5,6]], header_row=[‘a’, ‘b’, ‘c’], frame=True)
sage: T
+------++------+
| a  | b  | c  |
+------++------+
| 1  | 2  | 3  |
+------++------+
| 4  | 5  | 6  |
+------++------+
sage: T.options(header_row=False)
sage: T
+------++------+
| a  | b  | c  |
+------++------+
```

(continues on next page)
If you do specify a list for `header_row`, an error is raised:

```python
sage: T.options(header_row=['x', 'y', 'z'])
Traceback (most recent call last):
...
TypeError: header_row should be either True or False.
```

The `transpose()` method

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:**

```python
sage: T = table([[1,2,3], [4,5,6]])
sage: T.transpose()
  1 4
  2 5
  3 6
```

```python
sage: T = table([[1,2,3], [4,5,6]], header_row=['x', 'y', 'z'], frame=True)
sage: T.transpose()
+---+---+---+
| x | 1 | 4 |
| y | 2 | 5 |
| z | 3 | 6 |
+---+---+---+
```

## 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 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
class sage.structure.sage_object.SageObject
```

```python
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('<sage>a</sage>')
(123)
sage: html.eval('<sage>a</sage>', locals={'a': 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:

```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'+
....='A09TXL0Y4OiAABJRU5ErkJggg==')
<iframe height="400" width="800"
src="http://data:image/png;base64,˓→iVBORw0KGgoAAAANSUhEUgAAAAUAAAAFCAYAAACNbyblAAAAHElEQVQI12P4//8/w38GIAXDIBKE0DHxgljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg=="></iframe>
```

```sage
class sage.misc.html.HtmlFragment
Bases: str, sage.structure.sage_object.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
```

4.2. HTML and MathML

289
sage: from sage.misc.html import HtmlFragment
sage: HtmlFragment('<b>test</b>')</sage>

_hat_extrepr_(display_manager, **kwds)

Rich Output Magic Method

See sage.repl.rich_output for details.

EXAMPLES:

name|sage: from sage.repl.rich_output import get_display_manager
name|sage: dm = get_display_manager()
name|sage: h = sage.misc.html.HtmlFragment('<b>old</b>')
name|sage: h._rich_repr_(dm)  # the doctest backend does not support html

OutputPlainText container
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)
<html>\[
\newcommand{\Bold}[1]{\mathbf{#1}}3\]<html>
sage: MathJax()(ZZ)
<html>
\[
\newcommand{\Bold}[1]{\mathbf{#1}}\Bold{Z}\]<html>

eval(x, globals=None, locals=None, mode='display', combine_all=False)

Render LaTeX input using MathJax. This returns a MathJaxExpr.

INPUT:

- x - a Sage object
- globals - a globals dictionary
- locals - extra local variables used when evaluating Sage code in x.
- mode - string (optional, default 'display'): 'display' for displaymath, 'inline' for inline math, or 'plain' for just the LaTeX code without the surrounding html and script tags.
- combine_all - boolean (Default: False): If combine_all is True and the input is a tuple, then it does not return a tuple and instead returns a string with all the elements separated by a single space.

OUTPUT:

A MathJaxExpr

EXAMPLES:

sage: from sage.misc.html import MathJax
sage: MathJax().eval(3, mode='display')
<html>\[
\newcommand{\Bold}[1]{\mathbf{#1}}3\]<html>
sage: MathJax().eval(3, mode='inline')
<html>
\[
\newcommand{\Bold}[1]{\mathbf{#1}}3\]<html>
sage: MathJax().eval(type(3), mode='inline')
<html>
\[
\newcommand{\Bold}[1]{\mathbf{#1}}\verb|<class|\phantom{\verb!x!}\verb|\rightarrow\verb|Sage.rings.integer.Integer>|\]<html>

(continues on next page)
class sage.misc.html.MathJaxExpr(y)
Bases: object
An arbitrary MathJax expression that can be nicely concatenated.

EXAMPLES:

```python
sage: from sage.misc.html import MathJaxExpr
sage: MathJaxExpr("a^2") + MathJaxExpr("x^{-1}")
a^2 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>\[
\newcommand{\Bold}[1]{\mathbf{#1}}
\frac{1}{2}\]
</html>
sage: html('<a href="http://sagemath.org">sagemath</a>')</a>
<sagemath</a>
sage: html('<a href="http://sagemath.org">sagemath</a>', strict=True)
<html>\[
\newcommand{\Bold}[1]{\mathbf{#1}}\verb|<a|\phantom{|\verb!x!}\verb|href=\n\rightarrow|"http://sagemath.org">sagemath</a>\]\</html>
```

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 $'. 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('This is \[2+2\].'))
This is \[2+2\].
sage: print(sage.misc.html.math_parse(r'\$2+2\$ is rendered to $2+2$.'))
<\span>$</span>2+2</span>$ is rendered to \(2+2\).
```

```python
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 (optional, 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'
```

## 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 \_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

\textbf{class} \texttt{sage.misc.latex.Latex(debug=False, slide=False, density=150, pdflatex=None, engine=None)}

\texttt{Bases: sage.misc.latex.LatexCall}

Enter, e.g.,

```
\%latex
The equation $y^2 = x^3 + x$ defines an elliptic curve.
We have $2006 = \texttt{sage\{factor\{2006\}\}}$.
```

in an input cell in the notebook to get a typeset version. Use \%latex\_debug to get debugging output.

Use \texttt{latex(\ldots)} to typeset a Sage object. Use \texttt{LatexExpr} to typeset LaTeX code that you create by hand.

Use \%slide instead to typeset slides.

\textbf{Warning:} You must have dvipng (or dvips and convert) installed on your operating system, or this command will not work.

\textbf{EXAMPLES:}

```sage
latex(x^20 + 1)
x^{20} + 1
latex(FiniteField(25, 'a'))
\Bold{F}_{5^{2}}
latex("hello")
\texttt{hello}
LatexExpr(r"\frac{x^2 - 1}{x + 1} = x - 1")
\frac{x^2 - 1}{x + 1} = x - 1
```

LaTeX expressions can be added; note that a space is automatically inserted:
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()
''
latex.add_macro("\newcommand{\foo}{bar}")
latex.extra_macros()''
latex.add_macro(""") # restore to default
```

add_package_to_preamble_if_available(package_name)

Adds a \usepackage{package_name} instruction to the latex preamble if not yet present there, and if package_name.sty is available in the LaTeX installation.

INPUT:

• package_name – a string

See also:

• add_to_preamble()
• has_file().

add_to_preamble(s)

Append to the string s of extra LaTeX macros, for use with %latex.

EXAMPLES:

```sage
latex.extra_preamble()
''
latex.add_to_preamble("\DeclareMathOperator{\Ext}{Ext}\usepackage{xypic}\")
```

At this point, a notebook cell containing

```latex
\Ext_A^*(\GF{2}, \GF{2}) \Rightarrow \pi_*^s*(S^0)
```

will be typeset correctly.

```sage
latex.add_to_preamble("\usepackage{xypic}\")
latex.extra_preamble()
'\DeclareMathOperator{\Ext}{Ext}\usepackage{xypic}'
```

Now one can put various xypic diagrams into a %latex cell, such as

```latex
\xymatrix{ \circ \ar[r]^{\r[d]^*\{a\} ` [rr]^\{b\} `/4pt[rr]^\{c\} ` [rrr]^\{d\}}
```

(continues on next page)
Reset the preamble to its default, the empty string:

```python
sage: latex.extra_preamble('')
sage: latex.extra_preamble()
''
```

blackboard_bold(t=\texttt{None})

Controls whether Sage uses blackboard bold or ordinary bold face for typesetting $\mathbb{Z}$, $\mathbb{R}$, etc.

**INPUT:**

- $t$ – boolean or \texttt{None}

**OUTPUT:**

If $t$ is \texttt{None}, return the current setting (\texttt{True} or \texttt{False}).

If $t$ is \texttt{True}, use blackboard bold ($\mathbb{\text{e}}$); otherwise use boldface ($\mathbf{\text{e}}$).

**EXAMPLES:**

```python
sage: latex.blackboard_bold()
False
sage: latex.blackboard_bold(True)
sage: latex.blackboard_bold()
True
sage: latex.blackboard_bold(False)
```

check_file(file_name, more_info='')

**INPUT:**

- `file_name` – a string
- `more_info` – a string (default: '')

Emit a warning if the local LaTeX installation does not include `file_name`. The string `more_info` is appended to the warning message. The warning is only emitted the first time this method is called.

**EXAMPLES:**

```python
sage: latex.check_file("article.cls")  # optional - latex
sage: latex.check_file("some_inexistent_file.sty")
Warning: `some_inexistent_file.sty` is not part of this computer's \TeX installation.
```

```python
sage: latex.check_file("some_inexistent_file.sty", "This file is required for \texttt{blah}. It can be downloaded from: http://blah.org/")
Warning: `some_inexistent_file.sty` is not part of this computer's \TeX installation.
This file is required for \texttt{blah}. It can be downloaded from: http://blah.org/
```

This test checks that the bug in trac ticket \#9091 is fixed:

```python
sage: latex.check_file("article.cls", "The article class is really critical.")  # optional - latex
```

4.3. LaTeX
engine\( (e=\text{None}) \)
Set Sage to use \( e \) as latex engine when typesetting with \texttt{view()}, in \%\texttt{latex} cells, etc.

\textbf{INPUT:}
- \( e \) – ‘latex’, ‘pdflatex’, ‘xelatex’ or \texttt{None}

If \( e \) is \texttt{None}, return the current engine.

If using the \texttt{XeLaTeX} engine, it will almost always be necessary to set the proper preamble with \texttt{extra_preamble()} or \texttt{add_to_preamble()}. For example:

\begin{verbatim}
latex.extra_preamble(r'''
\usepackage{fontspec,xunicode,xltxtra}
\setmainfont[Mapping=tex-text]{some font here}
\setmonofont[Mapping=tex-text]{another font here}''')
\end{verbatim}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: latex.engine()
'pdflatex'
sage: latex.engine("latex")
sage: latex.engine()
'latex'
sage: latex.engine("xelatex")
sage: latex.engine()
'xelatex'
\end{verbatim}

eval\( (x, \text{globals}, \text{strip}=\text{False}, \text{filename}=\text{None}, \text{debug}=\text{None}, \text{density}=\text{None}, \text{pdflatex}=\text{None}, \text{engine}=\text{None}, \text{locals}=\{\}) \)
Compile the formatted tex given by \( x \) as a png and writes the output file to the directory given by \texttt{filename}.

\textbf{INPUT:}
- \texttt{globals} – a globals dictionary
- \( x \) – string to evaluate.
- \texttt{strip} – ignored
- \texttt{filename} – output filename
- \texttt{debug} – whether to print verbose debugging output
- \texttt{density} – how big output image is.
- \texttt{pdflatex} – whether to use pdflatex. This is deprecated. Use \texttt{engine} option instead.
- \texttt{engine} – latex engine to use. Currently latex, pdflatex, and xelatex are supported.
- \texttt{locals} - extra local variables used when evaluating Sage code in \( x \).

\textbf{Warning:} When using latex (the default), you must have ‘dvipng’ (or ‘dvips’ and ‘convert’) installed on your operating system, or this command will not work. When using pdflatex or xelatex, you must have ‘convert’ installed.

\textbf{OUTPUT:}
If it compiled successfully, this returns an empty string ‘’ , otherwise it returns \texttt{None}.

\textbf{EXAMPLES:}

```python
sage: fn = tmp_filename()
sage: latex.eval("$\ZZ[x]$", locals(), filename=fn)  # not tested
''
sage: latex.eval(r"\ThisIsAnInvalidCommand", {})  # optional -- ImageMagick
An error occurred...
No pages of output...
```

### extra_macros

**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}"")
sage: latex.extra_macros()  # current string
''
sage: latex.extra_macros(""")
sage: latex.extra_macros()  # current string
''
```

### extra_preamble

**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("\DeclareMathOperator{\Ext}{Ext}"")
sage: latex.extra_preamble()  # current string
''
sage: latex.extra_preamble("\usepackage{fontspec,xunicode,xltxtra}\setmainfont[Mapping=tex-text]{UnBatang}\setmonofont[Mapping=tex-text]{UnDotum}"")
sage: latex.extra_preamble()  # current string
''
sage: latex.extra_preamble(""")
sage: latex.extra_preamble()  # current string
''
```

### has_file

**INPUT:**

- file_name – a string

4.3. LaTeX
Tests whether the local LaTeX installation includes file_name.

EXAMPLES:

```
sage: latex.has_file("article.cls")       # optional - latex
True
sage: latex.has_file("some_inexistent_file.sty")
False
```

**matrix_column_alignment(align=None)**
Changes the column-alignment of the LaTeX representation of matrices.

**INPUT:**

- **align** - a string ('r' for right, 'c' for center, 'l' for left) or None.

**OUTPUT:**

If align is None, then returns the current alignment-string. Otherwise, set this alignment.

The input align can be any string which the LaTeX array-environment understands as a parameter for aligning a column.

**EXAMPLES:**

```
sage: a = matrix(1, 1, [42])
sage: latex(a)
\left(\begin{array}{r}
  42
\end{array}\right)
sage: latex.matrix_column_alignment('c')
sage: latex(a)
\left(\begin{array}{c}
  42
\end{array}\right)
sage: latex.matrix_column_alignment('l')
sage: latex(a)
\left(\begin{array}{l}
  42
\end{array}\right)
sage: latex.matrix_column_alignment('r')
```

Restore defaults:

```
sage: latex.matrix_column_alignment('r')
```

**matrix_delimiters(left=None, right=None)**
Change the left and right delimiters for the LaTeX representation of matrices

**INPUT:**

- **left, right** - strings or None

If both left and right are None, then return the current delimiters. Otherwise, set the left and/or right delimiters, whichever are specified.

Good choices for left and right are any delimiters which LaTeX understands and knows how to resize; some examples are:

- parentheses: ‘(’, ‘)’
- brackets: ‘[‘, ‘]’
• braces: ‘{}, ‘\}\’
• vertical lines: ‘|’
• angle brackets: ‘\langle’, ‘\rangle’

Note: Putting aside aesthetics, you may combine these in any way imaginable; for example, you could set left to be a right-hand bracket ‘]’ and right to be a right-hand brace ‘\}', and it will be typeset correctly.

EXAMPLES:

```
sage: a = matrix(1, 1, [17])
sage: latex(a)
\left(\begin{array}{r}
  17
\end{array}\right)
sage: latex.matrix_delimiters("[", "]")
sage: latex(a)
\left[\begin{array}{r}
  17
\end{array}\right]
sage: latex.matrix_delimiters(left="\{")
sage: latex(a)
\left\{\begin{array}{r}
  17
\end{array}\right\}
sage: latex.matrix_delimiters()
[‘\\{‘, ‘\}’]
```

Restore defaults:

```
sage: latex.matrix_delimiters("(", ")")
```

`vector_delimiters(left=None, right=None)`

Change the left and right delimiters for the LaTeX representation of vectors

INPUT:

• left, right – strings or None

If both left and right are None, then return the current delimiters. Otherwise, set the left and/or right delimiters, whichever are specified.

Good choices for left and right are any delimiters which LaTeX understands and knows how to resize; some examples are:

• parentheses: ‘(, ‘)’
• brackets: ‘[’, ‘]’
• braces: ‘\{’, ‘\}’
• vertical lines: ‘|’
• angle brackets: ‘\langle’, ‘\rangle’

Note: Putting aside aesthetics, you may combine these in any way imaginable; for example, you could set
left to be a right-hand bracket ‘]’ and right to be a right-hand brace ‘\}', and it will be typeset correctly.

EXAMPLES:

```python
sage: a = vector(QQ, [1,2,3])
sage: latex(a)
\left(1,\,2,\,3\right)
sage: latex(vector_delimiters("[", "]")
sage: latex(a)
\left[1,\,2,\,3\right]
sage: latex(vector_delimiters(right="\}\))
sage: latex(a)
\left[1,\,2,\,3\right]\}
sage: latex(vector_delimiters() ['\', '\\'])
```

Restore defaults:

```python
sage: 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 Latex inherits from this. This class is used in latex_macros, while functions from latex_macros are used in Latex, so this is here primarily to avoid circular imports.

EXAMPLES:

```python
sage: from sage.misc.latex import LatexCall
sage: LatexCall()(ZZ)
\Bold{Z}
sage: LatexCall().__call__(ZZ)
\Bold{Z}
```

This returns an instance of the class LatexExpr:

```python
sage: type(LatexCall()(ZZ))
<class 'sage.misc.latex.LatexExpr'>
```

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/:}
```

(continues on next page)
class diagram
   Bases: sage.structure.sage_object.SageObject

   LaTeX example for testing display of commutative diagrams. See its string representation for details.

   EXAMPLES:

   sage: from sage.misc.latex import latex_examples
   sage: CD = latex_examples.diagram()
   sage: CD
   LaTeX example for testing display of a commutative diagram...

class graph
   Bases: sage.structure.sage_object.SageObject

   LaTeX example for testing display of graphs. See its string representation for details.

   EXAMPLES:

   sage: from sage.misc.latex import latex_examples
   sage: G = latex_examples.graph()
   sage: G
   LaTeX example for testing display of graphs...

class knot
   Bases: sage.structure.sage_object.SageObject

   LaTeX example for testing display of knots. See its string representation for details.

   EXAMPLES:

   sage: from sage.misc.latex import latex_examples
   sage: K = latex_examples.knot()
   sage: K
   LaTeX example for testing display of a knot...

class pstricks
   Bases: sage.structure.sage_object.SageObject

   LaTeX example for testing display of pstricks output. See its string representation for details.

   EXAMPLES:

   sage: from sage.misc.latex import latex_examples
   sage: PS = latex_examples.pstricks()
   sage: PS
   LaTeX example for testing display of pstricks...

class sage.misc.latex.LatexExpr
   Bases: str

   A class for LaTeX expressions.
Normally, objects of this class are created by a \texttt{latex()} call. It is also possible to generate \texttt{LatexExpr} directly from a string, which must contain valid LaTeX code for typesetting in math mode (without dollar signs). In the Sage notebook, use \texttt{pretty_print()} or the “Typeset” checkbox to actually see the typeset LaTeX code; alternatively, from either the command-line or the notebook, use the \texttt{view()} function.

\textbf{INPUT:}

- \texttt{str} – a string with valid math mode LaTeX code (or something which can be converted to such a string).

\textbf{OUTPUT:}

- \texttt{LatexExpr} wrapping the string representation of the input.

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: latex(x^20 + 1)}
\texttt{x^{20} + 1}
\texttt{sage: LatexExpr(r"\frac{x^2 + 1}{x - 2}")}
\texttt{\frac{x^2 + 1}{x - 2}}
\end{verbatim}

\texttt{LatexExpr} simply converts to string without doing anything extra, it does \textit{not} call \texttt{latex()}:

\begin{verbatim}
\texttt{sage: latex(ZZ)}
\texttt{\Bold{Z}}
\texttt{sage: LatexExpr(ZZ)}
\texttt{Integer Ring}
\end{verbatim}

The result of \texttt{latex()} is of type \texttt{LatexExpr}:

\begin{verbatim}
\texttt{sage: L = latex(x^20 + 1)}
\texttt{sage: L}
\texttt{x^{20} + 1}
\texttt{sage: type(L)}
\texttt{<class 'sage.misc.latex.LatexExpr'>}
\end{verbatim}

A \texttt{LatexExpr} can be converted to a plain string:

\begin{verbatim}
\texttt{sage: str(latex(x^20 + 1))}
\texttt{'x^{20} + 1'}
\end{verbatim}

\texttt{sage.misc.latex.None_function(x)}

Returns the LaTeX code for \texttt{None}.

\textbf{INPUT:}

- \texttt{x} – \texttt{None}

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: from sage.misc.latex import None_function}
\texttt{sage: print(None_function(None))}
\texttt{\mathrm{None}}
\end{verbatim}

\texttt{sage.misc.latex.bool_function(x)}

Returns the LaTeX code for a boolean \texttt{x}.

\textbf{INPUT:}

- \texttt{x} – boolean
EXAMPLES:

```python
sage: from sage.misc.latex import bool_function
sage: print(bool_function(2==3))
\text{False}
sage: print(bool_function(3==(2+1)))
\text{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](http://docs.python.org/library/constants.html)

EXAMPLES:

```python
sage: from sage.misc.latex import builtin_constant_function
sage: builtin_constant_function(True)
'\text{True}'
sage: builtin_constant_function(None)
'\text{None}'
sage: builtin_constant_function(NotImplemented)
'\text{NotImplemented}'
sage: builtin_constant_function(Ellipsis)
'\text{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)'
```

`sage.misc.latex.dict_function(x)`

Return the LaTeX code for a dictionary `x`.

INPUT:

- `x` – a dictionary

EXAMPLES:
```python
sage: from sage.misc.latex import dict_function
sage: x,y,z = var('x,y,z')
sage: print(dict_function({x/2: y^2}))
\left\{\frac{1}{2} \, x : y^{2}\right\}
sage: d = {(1,2,x^2): [sin(z^2), y/2]}
sage: latex(d)
\left\{(1, 2, x^{2}) : \left[\sin(z^{2}), \frac{1}{2} \, y\right]\right\}
```

**sage.misc.latex.float_function**

Returns the LaTeX code for a python float \( x \).

**INPUT:**

- \( x \) – a python float

**EXAMPLES:**

```python
sage: from sage.misc.latex import float_function
sage: float_function(float(3.14))
3.14
sage: float_function(float(1e-10))
1 \times 10^{-10}
```

**sage.misc.latex.has_latex_attr**

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))
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: T = type(identity_matrix(3)); T
<type 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
sage: hasattr(T, '_latex_')
True
sage: T._latex_()
Traceback (most recent call last):
... TypeError: ..._latex..... needs an argument
sage: has_latex_attr(T)
False
```

**sage.misc.latex.have_convert**

Return True if this computer has the program convert.

If this computer does not have convert installed, you may obtain it (along with the rest of the ImageMagick suite) from http://www.imagemagick.org

**EXAMPLES:**
sage.misc.latex.have_dvipng()
Return True if this computer has the program dvipng.
If this computer does not have dvipng installed, you may obtain it from http://sourceforge.net/projects/dvipng/

EXAMPLES:

```python
sage: from sage.misc.latex import have_dvipng
sage: have_dvipng() # random
True
```

sage.misc.latex.have_latex()
Return True if this computer has the program latex.
If this computer does not have LaTeX installed, you may obtain it from http://ctan.org/.

EXAMPLES:

```python
sage: from sage.misc.latex import have_latex
sage: have_latex() # random
True
```

sage.misc.latex.have_pdflatex()
Return True if this computer has the program pdflatex.
If this computer does not have pdflatex installed, you may obtain it from http://ctan.org/.

EXAMPLES:

```python
sage: from sage.misc.latex import have_pdflatex
sage: have_pdflatex() # random
True
```

sage.misc.latex.have_xelatex()
Return True if this computer has the program xelatex.
If this computer does not have xelatex installed, you may obtain it from http://ctan.org/.

EXAMPLES:

```python
sage: from sage.misc.latex import have_xelatex
sage: have_xelatex() # random
True
```

sage.misc.latex.latex(x, combine_all=False)
Return a LatexExpr built out of the argument x.

INPUT:

- x – a Sage object
- combine_all – boolean (Default: False) If combine_all is True and the input is a tuple, then it does not return a tuple and instead returns a string with all the elements separated by a single space.

OUTPUT:

A LatexExpr built from x
EXAMPLES:

```python
sage: latex(Integer(3))  # indirect doctest
3
sage: latex(1==0)
\text{\textbackslash mathrm{False}}

sage: print(latex([x, 2]))
\left[x, 2\right]
```

Check that trac ticket #11775 is fixed:

```python
sage: latex((x, 2), combine_all=True)
x 2
```

`sage.misc.latex.latex_extra_preamble()`

Return the string containing the user-configured preamble, `sage_latex_macros`, and any user-configured macros. This is used in the `eval()` method for the `Latex` class, and in `_latex_file_()`: it follows either `LATEX_HEADER` or `SLIDE_HEADER` (defined at the top of this file) which is a string containing the documentclass and standard usepackage commands.

EXAMPLES:

```python
sage: from sage.misc.latex import latex_extra_preamble
sage: print(latex_extra_preamble())
\newcommand{\ZZ}{\textbf{\textbackslash Z}}
\newcommand{\NN}{\textbf{\textbackslash N}}
\newcommand{\RR}{\textbf{\textbackslash R}}
\newcommand{\CC}{\textbf{\textbackslash C}}
\newcommand{\QQ}{\textbf{\textbackslash Q}}
\newcommand{\QQbar}{\overline{\textbf{\textbackslash Q}}}
\newcommand{\GF}{\textbf{\textbackslash F}_{#1}}
\newcommand{\Zp}{\textbf{\textbackslash Z}_{#1}}
\newcommand{\Qp}{\textbf{\textbackslash Q}_{#1}}
\newcommand{\Zmod}{\textbf{\textbackslash Z}/#1}\textbf{\textbackslash Z}
\newcommand{\CDF}{\textbf{\textbackslash C}}
\newcommand{\CIF}{\textbf{\textbackslash I} \textbf{\textbackslash R}}
\newcommand{\RLF}{\textbf{\textbackslash R}}
\newcommand{\Bold}{\mathbf{#1}}
```

`sage.misc.latex.latex_variable_name(x, is_fname=False)`

Return latex version of a variable name.

Here are some guiding principles for usage of this function:

1. If the variable is a single letter, that is the latex version.
2. If the variable name is suffixed by a number, we put the number in the subscript.
3. If the variable name contains an '_' we start the subscript at the underscore. Note that #3 trumps rule #2.
4. If a component of the variable is a Greek letter, escape it properly.
5. Recurse nicely with subscripts.
Refer to the examples section for how these rules might play out in practice.

EXAMPLES:

```
sage: from sage.misc.latex import latex_variable_name
sage: latex_variable_name('a')
'a'
sage: latex_variable_name('abc')
'\mathit{abc}'
sage: latex_variable_name('sigma')
'\sigma'
sage: latex_variable_name('sigma_k')
'\sigma_{k}'
sage: latex_variable_name('sigma389')
'\sigma_{389}'
sage: latex_variable_name('beta_00')
'\beta_{00}'
sage: latex_variable_name('Omega84')
'\Omega_{84}'
sage: latex_variable_name('sigma_alpha')
'\sigma_{\alpha}'
sage: latex_variable_name('nothing1')
'\mathit{nothing}_{1}'
sage: latex_variable_name('nothing1', is_fname=True)
'{\rm nothing}_{1}'
sage: latex_variable_name('nothing_abc')
'\mathit{nothing}_{\mathit{abc}}'
sage: latex_variable_name('nothing_abc', is_fname=True)
'{\rm nothing}_{{\rm abc}}'
sage: latex_variable_name('alpha_beta_gamma12')
'\alpha_{\beta_{\gamma_{12}}}'
sage: latex_variable_name('x_ast')
'x_{\ast}'
```

```
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 “\{a\}” or “\mbox{a}” if “is_fname” flag is True or False.

INPUT:
• a – string

OUTPUT:
A string

EXAMPLES:
```
sage: from sage.misc.latex import latex_varify
sage: latex_varify('w')
'w'
sage: latex_varify('aleph')
'\mathit{aleph}'
sage: latex_varify('aleph', is_fname=True)
'{\rm aleph}'
sage: latex_varify('alpha')
```
```
sage.misc.latex.list_function(x)

Returns the LaTeX code for a list x.

INPUT: x - a list

EXAMPLES:

```
sage: from sage.misc.latex import list_function
sage: list_function([1,2,3])
'\left[1, 2, 3\right]'
sage: latex([[1,2,3]]) # indirect doctest
\left[\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)
```

sage.misc.latex.png(x, filename, density=150, debug=False, do_in_background=False, tiny=False, pdflatex=True, engine='pdflatex')

Create a png image representation of x and save to the given filename.

INPUT:

- x – object to be displayed
- filename – file in which to save the image
- density – integer (default: 150)
- debug – bool (default: False): print verbose output
- do_in_background – bool (default: False): Unused, kept for backwards compatibility
- tiny – bool (default: False): use ‘tiny’ font
- pdflatex – bool (default: True): use pdflatex. This option is deprecated. Use engine option instead. See below.
- engine – (default: 'pdflatex') 'latex', 'pdflatex', or 'xelatex'

EXAMPLES:

```
sage: from sage.misc.latex import png
sage: png(ZZ[x], os.path.join(SAGE_TMP, "zz.png")) # random - error if no latex
```

sage.misc.latex.repr_lincomb(symbols, coeffs)

Compute a latex representation of a linear combination of some formal symbols.

INPUT:
• symbols – list of symbols
• coeffs – list of coefficients of the symbols

OUTPUT:
A string

EXAMPLES:

```
sage: t = PolynomialRing(QQ, 't').0
data: from sage.misc.latex import repr_lincomb
sage: repr_lincomb(['a', 's', 't'], [-t, t - 2, t^12 + 2])
'-t\text{\texttt{a}} + \left(t - 2\right)\text{\texttt{s}} + \left(t^{12} + 2\right)'
sage: repr_lincomb(['a', 'b'], [1,1])
'\text{\texttt{a}} + \text{\texttt{b}}'
```

Verify that a certain corner case works (see trac ticket #5707 and trac ticket #5766):

```
sage: repr_lincomb([1,5,-3],[2,8/9,7])
'2\cdot 1 + \frac{8}{9}\cdot 5 + 7\cdot -3 '
```

Verify that trac ticket #17299 (latex representation of modular symbols) is fixed:

```
sage: x = EllipticCurve('64a1').modular_symbol_space(sign=1).basis()[0]
sage: from sage.misc.latex import repr_lincomb
sage: latex(x.modular_symbol_rep())
\left\{
\frac{-3}{11}, \frac{-1}{4}\right\} - \left\{
\frac{3}{13}, \frac{1}{4}\right\}
```

Verify that it works when the symbols are numbers:

```
sage: x = FormalSum([(1,2),(3,4)])
sage: latex(x)
2 + 3\cdot 4
```

Verify that it works when `bv in CC` raises an error:

```
sage: x = FormalSum([(1,'x'),(2,'y')])
sage: latex(x)
\text{\texttt{x}} + 2\text{\texttt{y}}
```

`sage.misc.latex.str_function(x)`

Return a LaTeX representation of the string `x`.

The main purpose of this function is to generate LaTeX representation for classes that do not provide a customized method.

If `x` contains only digits with, possibly, a single decimal point and/or a sign in front, it is considered to be its own representation. Otherwise each line of `x` is wrapped in a `\texttt{command` and these lines are assembled in a left-justified array. This gives to complicated strings the closest look to their “terminal representation”.

**Warning:** Such wrappers cannot be used as arguments of LaTeX commands or in command definitions. If this causes you any problems, they probably can be solved by implementing a suitable `_latex_` method for an appropriate class.
INPUT:

• \( x \) – a string.

OUTPUT:

A string

EXAMPLES:

```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{hello}\ _\text{world}''
sage: str_function('-1.00000?') # trac 12178
'-1.00000?'
```

`sage.misc.latex.tuple_function(x, combine_all=False)`

Returns the LaTeX code for a tuple \( x \).

INPUT:

• \( x \) – a tuple

• \( combine\_all \) – boolean (Default: False) If \( combine\_all \) is True, then it does not return a tuple and instead returns a string with all the elements separated by a single space. It does not collapse tuples which are inside tuples.

EXAMPLES:

```python
sage: from sage.misc.latex import tuple_function
sage: tuple_function((1,2,3))
'\left(1, 2, 3\right)'
sage: tuple_function((1,2,3), combine_all=True)
'1 2 3'
sage: tuple_function(((1,2),3), combine_all=True)
'\left(1, 2\right) 3'
```

`sage.misc.latex.view(objects, title='Sage', debug=False, sep=' ', tiny=False, pdflatex=None, engine=None, viewer=None, tightpage=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
• \texttt{sep} – string (default: ''): separator between math objects

• \texttt{tiny} – bool (default: False): use tiny font.

• \texttt{pdflatex} – bool (default: False): use \texttt{pdflatex}. This is deprecated. Use 'engine' option instead.

• \texttt{engine} – string or \texttt{None} (default: \texttt{None}). Can take the following values:
  – \texttt{None} – the value defined in the \LaTeX global preferences \texttt{latex.engine()} is used.
  – 'pdflatex' – compilation does \texttt{tex} -> \texttt{pdf}
  – 'xelatex' – compilation does \texttt{tex} -> \texttt{pdf}
  – 'latex' – compilation first tries \texttt{tex} -> \texttt{dvi} -> \texttt{png} and if an error occurs then tries \texttt{dvi} -> \texttt{ps} -> \texttt{pdf}.

This is slower than 'pdflatex' and known to be broken when overfull hbox are detected.

• \texttt{viewer} – string or \texttt{None} (default: \texttt{None}): specify a viewer to use; currently the only options are \texttt{None} and 'pdf'.

• \texttt{tightpage} – bool (default: True): use the \LaTeX package 'preview' with the 'tightpage' option.

• \texttt{margin} – float or \texttt{None} (default: \texttt{None}): adds a margin of \texttt{margin} mm; has no affect if the option \texttt{tightpage} is \texttt{False}.

• \texttt{mode} – string (default: 'inline'): 'display' for displaymath or 'inline' for inline math

• \texttt{combine_all} – bool (default: False): If \texttt{combine_all} is \texttt{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:}

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 \texttt{objects} is typeset on its own line, with the string \texttt{sep} inserted between these lines.

The value of \texttt{sep} is inserted between each element of the list \texttt{objects}; you can, for example, add vertical space between objects with \texttt{sep='}\\vspace{15mm}'', while \texttt{sep='}\\hrule' adds a horizontal line between objects, and \texttt{sep='}\\newpage' inserts a page break between objects.

If \texttt{pdflatex} is \texttt{True}, then the latex engine is set to \texttt{pdflatex}.

If the \texttt{engine} is either \texttt{pdflatex} or \texttt{xelatex}, it produces a pdf file. Otherwise, it produces a dvi file, and if the program \texttt{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 \texttt{viewer} to 'pdf' forces the use of a separate viewer, even in notebook mode. This also sets the latex engine to be \texttt{pdflatex} if the current engine is latex.

Setting the option \texttt{tightpage} to \texttt{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 \texttt{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, \texttt{tightpage} adds

\begin{verbatim}
\usepackage[tightpage,active]{preview}
\PreviewEnvironment{page}
\end{verbatim}

\texttt{to the La\TeX preamble, and replaces the '}' and ']' around each object by 'begin{page}$ and '$end{page}'. Setting \texttt{tightpage} to \texttt{False} turns off this behavior and provides the latex output as a full page. If \texttt{tightpage} is set to \texttt{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.docs.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. The list of MathJax macros is used in the file sagenb/notebook/tutorial.py to define MathJax macros for use in the live documentation (and also in the notebook).

Any macro defined here may be used in docstrings or in the tutorial (or other pieces of documentation). In a docstring, for example, “ZZ” in backquotes (demarking math mode) will appear as “ZZ” in interactive help, but will be typeset as “\Bold{Z}” in the reference manual.

More details on the list macros: the entries are lists or tuples of the form [name] or [name, arguments], where name is a string and arguments consists of valid arguments for the Sage object named name. For example, ["ZZ"] and ["GF", 2] produce the LaTeX macros ‘newcommand\{ZZ\}{\Bold{Z}}’ and ‘newcommand\{GF\}[1]{\Bold{F}_{#1}}’, respectively. (For the second of these, latex(GF(2)) is called and the string ‘2’ gets replaced by ‘#1’, so ["GF", 17] would have worked just as well. ["GF", p] would have raised an error, though, because p is not defined, and ["GF", 4] would have raised an error, because to define the field with four elements in Sage, you also need to specify the name of a generator.)

To see evidence of the results of the code here, run sage --docbuild tutorial latex (for example), and look at the resulting LaTeX file in SAGE_DOC/latex/en/tutorial/. The preamble should contain ‘newcommand’ lines for each of the entries in macros.

sage.misc.latex_macros.convert_latex_macro_to_mathjax(macro)
This converts a LaTeX macro definition (newcommand...) to a MathJax macro definition (MathJax.Macro...).

INPUT:

- macro - LaTeX macro definition

See the web page http://www.mathjax.org/docs/1.1/options/TeX.html for a description of the format for MathJax macros.

EXAMPLES:

```
sage: from sage.misc.latex_macros import convert_latex_macro_to_mathjax
sage: convert_latex_macro_to_mathjax('\\newcommand{\ZZ}{\Bold{Z}}')
'ZZ: "\\\Bold{Z}"'
sage: convert_latex_macro_to_mathjax('\\newcommand{\GF}[1]{\Bold{F}_{#1}}')
'GF: ["\\\Bold{F}_{#1}",1]'
```

sage.misc.latex_macros.produce_latex_macro(name, *sample_args)
Produce a string defining a LaTeX macro.

INPUT:

- name - name of macro to be defined, also name of corresponding Sage object
- sample_args - (optional) sample arguments for this Sage object

EXAMPLES:
sage: from sage.misc.latex_macros import produce_latex_macro
sage: produce_latex_macro('ZZ')
'\newcommand{\ZZ}{\Bold{Z}}'

If the Sage object takes arguments, then the LaTeX macro will accept arguments as well. You must pass valid arguments, which will then be converted to #1, #2, etc. in the macro definition. The following allows the use of “GF{p^n}”, for example:

sage: produce_latex_macro('GF', 37)
'\newcommand{\GF}{\Bold{F}_{#1}}'

If the Sage object is not in the global name space, describe it like so:

sage: produce_latex_macro('sage.rings.finite_rings.finite_field_constructor.˓→FiniteField', 3)
'\newcommand{\FiniteField}{\Bold{F}_{#1}}'

sage.misc.latex_macros.sage_latex_macros()
Return list of LaTeX macros for Sage. This just runs the function produce_latex_macro() on the list macros defined in this file, and appends sage_configurable_latex_macros. To add a new macro for permanent use in Sage, modify macros.

EXAMPLES:

sage: from sage.misc.latex_macros import sage_latex_macros
sage: sage_latex_macros()
[\newcommand{\ZZ}{\Bold{Z}}, \newcommand{\NN}{\Bold{N}}, ...]

sage.misc.latex_macros.sage_mathjax_macros()
Return list of MathJax macro definitions for Sage as JavaScript. This feeds each item output by sage_latex_macros() to convert_latex_macro_to_mathjax().

EXAMPLES:

sage: from sage.misc.latex_macros import sage_mathjax_macros
sage: sage_mathjax_macros()
[ZZ: "\Bold{Z}", NN: "\Bold{N}", ...]
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.

```python
class sage.misc.persist.SagePickler(file_obj, persistent_id=None, py2compat=True)
Bases: sage.misc.persist._BasePickler

Subclass pickle.Pickler with Sage-specific default options, and built-in support for external object persistence.

INPUT:
- `file_obj` – a readable file-like object returning bytes from which the pickle data will be loaded.
- `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.
- `py2compat` – on Python 3 only, this creates pickle files 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 `pickle.load()`, where in Python-3 Sage sets the default encoding='latin1'. This is essential to make pickles readable in Python-3 that were created in Python-2. See trac ticket #28444 for details.

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
```

The `SagePickler` can also be passed a `persistent_id` 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
....:     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:

```python
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: sage.misc.persist._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
....:     return table[obj_id]
```

classmethod loads(data, **kwargs)
Equivalent to pickle.dumps() but using the sage.misc.persist.SagePickler.

INPUT:
• data - the pickle data as bytes.
• kwargs - keyword arguments passed to the \texttt{sage.misc.persist.SageUnpickler} constructor.

OUTPUT:
• obj - the object that was serialized to the given pickle data.

EXAMPLES:

\begin{verbatim}
sage: import pickle
gherkin = pickle.dumps(1)
sage: SageUnpickler.loads(gherkin)
1
\end{verbatim}

\texttt{sage.misc.persist.db(name)}
Load object with given name from the Sage database. Use \texttt{x.db(name)} or \texttt{db_save(x, name)} to save objects to the database.

The database directory is $\texttt{HOME/.sage/db}$.

\texttt{sage.misc.persist.db_save(x, name=None)}
Save x to the Sage database.

The database directory is $\texttt{HOME/.sage/db}$.

\texttt{sage.misc.persist.dumps(obj, compress=True)}
Dump obj to a string s. To recover obj, use \texttt{loads(s)}.

See also:
\texttt{loads()}

EXAMPLES:

\begin{verbatim}
sage: a = 2/3
sage: s = dumps(a)
sage: a2 = loads(s)
sage: type(a) is type(a2)
True
sage: a2
2/3
\end{verbatim}

\texttt{sage.misc.persist.load(compress=True, verbose=True, *filename, **kwargs)}
Load Sage object from the file with name filename, which will have an \texttt{.sobj} extension added if it doesn’t have one. Or, if the input is a filename ending in \texttt{.py, .pyx, .sage, .spyx, .f, .f90} or \texttt{.m}, load that file into the current running session.

Loaded files are not loaded into their own namespace, i.e., this is much more like Python’s \texttt{execfile} than Python’s \texttt{import}.

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

When a pickle created with Python 2 is unpickled in Python 3, Sage uses the default encoding \texttt{latin1} to unpickle data of type \texttt{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 = 'http://www.sagemath.org/files/test.sobj'  # optional - internet
sage: s = load(u)                                    # optional - internet
Attempting to load remote file: http://www.sagemath.org/files/test.sobj
Loading started
Loading ended
sage: s                                             # optional - internet
'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]
```

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
```

We can load Fortran files:

```python
sage: code = '      subroutine hello
               print *, "Hello World!"
      end subroutine hello'

sage: t = tmp_filename(ext=".F")

sage: with open(t, 'w') as f:
    ....:     _ = f.write(code)

sage: load(t)

sage: hello
<fortran object>
```

```python
sage.misc.persist.load_sage_element(cls, parent, dic_pic)
sage.misc.persist.load_sage_object(cls, dic)
```

5.2. Object persistence 319
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])
sage: s = dumps(a)
sage: loads(s)
[ 1  2]
[ 3 -4/3]
```

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

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 trac ticket #28444 for details.

```python
sage: class Foo(object):
    ...:     def __init__(self, s):
    ...:         self.bar = s
    ...:     def __reduce__(self):
    ...:         return Foo, (self.bar,)
    ...:
sage: import __main__
sage: __main__.Foo = Foo
```

The data that is passed to loads in the following line was created by dumps(Foo('\\x80\x97')) in Python-2.

```python
sage: g = loads(b'\x9c\x80\x89\x8f\xc8\xe7\x7f\xe7\x0d\x07\x0c\x00\x89\xc5\x0f\x08', encoding='ASCII') #py3
sage: type(g), g.bar
(<class '__main__.Foo'>, '\x80\x97')
```

The following line demonstrates what would happen without trac ticket #28444:

```python
sage: loads(b'\x9c\x80\x89\x8f\xc8\xe7\x7f\xe7\x0d\x07\x0c\x00\x89\xc5\x0f\x08', encoding='ASCII') #py3
Traceback (most recent call last):
  ... UnicodeDecodeError: 'ascii' codec can't decode byte 0x80 in position 0: ordinal...
```

(continues on next page)
sage.misc.persist.make_None(*args, **kwds)
Do nothing and return None. Used for overriding pickles when that pickle is no longer needed.

EXAMPLES:

```python
sage: from sage.misc.persist import make_None
sage: print(make_None(42, pi, foo='bar'))
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:

```python
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
```

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:

```python
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
```

After following the advice from the error message, unpickling works:

```python
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.

```python
object.__setstate__(state)
```

Upon unpickling, if the class also defines the method `__setstate__`, it is called with the unpickled state. If there is no `__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 `__getstate__` and `__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:

```python
sage: from sage.structure.element import Element
c sage: class SourPickle(CombinatorialObject): pass
c sage: class SweetPickle(CombinatorialObject, Element): pass
c sage: import __main__
c sage: __main__.SourPickle = SourPickle
c sage: __main__.SweetPickle = SweetPickle
c sage: gherkin = dumps(SourPickle([1, 2, 3]))
```

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

```python
sage: from sage.misc.persist import register_unpickle_override
c sage: register_unpickle_override('__main__', 'SourPickle', SweetPickle)
c sage: loads(gherkin)
Traceback (most recent call last):
 ... KeyError: 0
```

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

```python
c 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)
[1, 2, 3]
sage: loads(dumps(SweeterPickle([1, 2, 3]))) # check that pickles work for
˓
SweeterPickle
[1, 2, 3]

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

```python
sage: class A(object):
...:     def __init__(self, value):
...:         self.original_attribute = value
...:     def __repr__(self):
...:         return 'A(%s)' % self.original_attribute
sage: class B(object):
...:     def __init__(self, value):
...:         self.new_attribute = value
...:     def __setstate__(self, state):
...:         try:
...:             self.new_attribute = state['new_attribute']
...:         except KeyError:  # an old pickle
...:             self.new_attribute = state['original_attribute']
...:     def __repr__(self):
...:         return 'B(%s)' % self.new_attribute
sage: import __main__
```

```
sage: __main__.A = A
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(A(10)))
B(10)
sage: loads(dumps(B(10)))
B(10)
```

Pickling for python classes and extension classes, such as cython, is different – again this is discussed in the

5.2. Object persistence 323
python pickling documentation. For the unpickling of extension classes you need to write a \_reduce\_() method which typically returns a tuple \((f, \text{args}, \ldots)\) such that \(f(*\text{args})\) 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
    #*'

    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:**

```sage
sage: a = matrix(2, [1,2,3,-5/2])
sage: objfile = os.path.join(SAGE_TMP, 'test.sobj')
sage: objfile_short = os.path.join(SAGE_TMP, 'test')
sage: save(a, objfile)
sage: load(objfile_short)
[ 1  2]
[ 3 -5/2]
sage: E = EllipticCurve([-1,0])
sage: P = plot(E)
sage: save(P, objfile_short)  # saves the plot to "test.sobj"
sage: save(P, filename=os.path.join(SAGE_TMP, "sage.png"), xmin=-2)
sage: save(P, os.path.join(SAGE_TMP, "filename.with.some.wrong.ext"))
```

(continues on next page)
Traceback (most recent call last):
...
ValueError: allowed file extensions for images are '.eps', '.pdf', '.pgf', '.png',
\x27.ps', '.sobj', '.svg'!
sage: print(load(objfile))
Graphics object consisting of 2 graphics primitives
sage: save("A python string", os.path.join(SAGE_TMP, 'test'))
sage: load(objfile)
'A python string'
sage: load(objfile_short)
'A python string'

sage.misc.persist.unpickle_all(dir, debug=False, run_test_suite=False)
Unpickle all sobj’s in the given directory, reporting failures as they occur. Also printed the number of successes and failure.

INPUT:

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

EXAMPLES:

sage: dir = tmp_dir()
sage: sage.misc.persist.picklejar('hello', dir)
sage: 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 unpickleOverride, register_unpickle_override
sage: unpickle_global('sage.rings.integer', 'Integer')
<type 'sage.rings.integer.Integer'>

Now we horribly break the pickling system:

sage: register_unpickle_override('sage.rings.integer', 'Integer', Rational, call_
\x27.name='sage.rings.rational', 'Rational')
sage: unpickle_global('sage.rings.integer', 'Integer')
<type 'sage.rings.rational.Rational'>

and we reach into the internals and put it back:
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
```

```python
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
```

## 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 ...>, ...)
sage: unpickle_function(pickle_function(foo))
<function foo at ...>
```

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:
EXAMPLES:

```python
sage: def f(N):
    return N+1
...

sage: g = pickle_function(f)
sage: h = unpickle_function(g)
sage: h(10)
11
```

```
sage.misc.fpickle.reduce_code(co)
EXAMPLES:

```
sage: def foo(N):
    return N+1
sage: sage.misc.fpickle.reduce_code(foo.__code__)
(<cyfunction code_ctor at ...>, ...)
```

```
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:

```
sage: def f(N,M):
    return N*M
...

sage: unpickle_function(pickle_function(f))(3,5)
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))
g_make_integer = unpickle_global('sage.rings.integer', 'make_integer')
g_make_integer('c1p')
sage: explain_pickle(dumps(polygen(QQ)))
g_Polynomial_rational_flint = unpickle_global('sage.rings.polynomial.polynomial_rational_flint', 'Polynomial_rational_flint')
g_unpickle_PolynomialRing = unpickle_global('sage.rings.polynomial.polynomial_ring_constructor', 'UnpicklePolynomialRing')
g_RationalField = unpickle_global('sage.rings.rational_field', 'RationalField')
```

(continues on next page)
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')
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**
  
  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:

```
from sage.rings.integer import integer
```

and if you ran `explain_pickle` in hypothetical future sage, it would generate:

```
from sage.rings.knuth_was_here import integer
```

but the current code wouldn’t work in the future sage.

If you pass the flag `default_assumptions=True`, then `explain_pickle` will generate code that would work in the absence of any special unpickling information. That is, in either current Sage or hypothetical future Sage, it would generate:

```
from sage.rings.integer import integer
```

The intention is that `default_assumptions` output is prettier (more human-readable), but may not actually work; so it is only intended for human reading.

There are several functions used in the output of `explain_pickle`. Here I give a brief description of what they usually do, as well as how to modify their operation (for instance, if you’re trying to get old pickles to work).

- `unpickle_global(module, classname)`: `unpickle_global(sage.foo.bar, ‘baz’)` is usually equivalent to `sage.foo.bar.baz`, but this can be customized with `register_unpickle_override`.
- `unpickle_newobj(klass, args)`: Usually equivalent to `klass.__new__(klass, *args)`. If `klass` is a Python class, then you can define `__new__()` to control the result (this result actually need not be an instance of `klass`). (This doesn’t work for Cython classes.)
- `unpickle_build(obj, state)`: If `obj` has a `__setstate__()` method, then this is equivalent to `obj.__setstate__(state)`. Otherwise uses `state` to set the attributes of `obj`. Customize by defining `__setstate__()`.
- `unpickle_instantiate(klass, args)`: Usually equivalent to `klass(*args)`. Cannot be customized.
- `unpickle_appends(lst, vals)`: Appends the values in `vals` to `lst`. If not `isinstance(lst, list)`, can be customized by defining a `append()` method.

```python
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 SageInputExpressions. We use this to help construct dictionary literals, instead of always starting with an empty dictionary and assigning to it.

class sage.misc.explain_pickle.PickleExplainer(sib, in_current_sage=False, default_assumptions=False, pedantic=False)
    Bases: object
    An interpreter for the pickle virtual machine, that executes symbolically and constructs SageInputExpressions instead of directly constructing values.
```
APPEND()
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_BININPUT(n)
MARK()
NEWFALSE()
NEWOBJ()
NEWTRUE()
NONE()
check_value(v)
Check that the given value is either a SageInputExpression or a PickleObject. Used for internal sanity checking.

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.check_value(7)
Traceback (most recent call last):
  ... AssertionError
sage: pe.check_value(sib(7))
```

is_mutable_pickle_object(v)
Test whether a PickleObject is mutable (has never been converted to a SageInputExpression).

EXAMPLES:

```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: v = PickleObject(1, sib(1))
sage: pe.is_mutable_pickle_object(v)
```
pop()
Pop a value from the virtual machine’s stack, and return it.

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(sib(7))
sage: pe.pop()
{atomic:7}
```

pop_to_mark()
Pop all values down to the ‘mark’ from the virtual machine’s stack, and return the values as a list.

EXAMPLES:

```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.push(sib(7))
sage: pe.push(sib('hello'))
sage: pe.pop_to_mark()
[{atomic:7}, {atomic:'hello'}]
```

push(v)
Push a value onto the virtual machine’s stack.

EXAMPLES:

```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(sib(7))
sage: pe.stack[-1]
{atomic:7}
```

push_and_share(v)
Push a value onto the virtual machine’s stack; also mark it as shared for sage_input if we are in pedantic mode.

EXAMPLES:
```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_and_share(sib(7))
{sage: pe.stack[-1]}
{atomic:7}
sage: pe.stack[-1]()._sie_share
True
```

### push_mark()

Push a ‘mark’ onto the virtual machine’s stack.

**EXAMPLES:**

```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'}
```

### share(v)

Mark a sage_input value as shared, if we are in pedantic mode.

**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: v = sib(7)
sage: v()._sie_share
False
```

(continues on next page)
class sage.misc.explain_pickle.PickleInstance(klass)
    Bases: object

    An object which can be used as the value of a PickleObject. Unlike other possible values of a PickleObject, a PickleInstance doesn't represent an exact value; instead, it gives the class (type) of the object.

class sage.misc.explain_pickle.PickleObject(value, expression)
    Bases: object

    Pickles have a stack-based virtual machine. The explain_pickle pickle interpreter mostly uses SageInputExpressions, from sage_input, as the stack values. However, sometimes we want some more information about the value on the stack, so that we can generate better (prettier, less confusing) code. In such cases, we push a PickleObject instead of a SageInputExpression. A PickleObject contains a value (which may be a standard Python value, or a PickleDict or PickleInstance), an expression (a SageInputExpression), and an “immutable” flag (which checks whether this object has been converted to a SageInputExpression; if it has, then we must not mutate the object, since the SageInputExpression would not reflect the changes).

class sage.misc.explain_pickle.TestAppendList
    Bases: list

    A subclass of list, with deliberately-broken append and extend methods. Used for testing explain_pickle.

    append()
    A deliberately broken append method.

    EXAMPLES:

    sage: from sage.misc.explain_pickle import *
    sage: v = TestAppendList()
    sage: v.append(7)  # py2
    Traceback (most recent call last):
    ...
    TypeError: append() takes exactly 1 argument (2 given)
    sage: v.append(7)  # py3
    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:

    sage: list.append(v, 7)
    sage: v
    [7]

extend()
    A deliberately broken extend method.

    EXAMPLES:

    sage: from sage.misc.explain_pickle import *
    sage: v = TestAppendList()
We can still extend by directly using the list method:

```
sage: list.extend(v, (3,1,4,1,5,9))
sage: v
[3, 1, 4, 1, 5, 9]
```

### sage.misc.explain_pickle.TestAppendNonlist

List-like class, carefully designed to test exact unpickling behavior. Used for testing explain_pickle.

### sage.misc.explain_pickle.TestBuild

A simple class with a `__getstate__` but no `__setstate__`. Used for testing explain_pickle.

### sage.misc.explain_pickle.TestBuildSetstate

A simple class with a `__getstate__` and a `__setstate__`. Used for testing explain_pickle.

### sage.misc.explain_pickle.TestGlobalFunnyName

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
```

### sage.misc.explain_pickle.TestGlobalNewName

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
```

### sage.misc.explain_pickle.TestGlobalOldName

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.
A featureless new-style class. When you try to unpickle an instance of this class, it is redirected to create a TestGlobalNewName instead. Used for testing explain_pickle.

EXAMPLES:

```python
sage: from sage.misc.explain_pickle import *
sage: loads(dumps(TestGlobalOldName()))
TestGlobalNewName
```

class sage.misc.explain_pickle.TestReduceGetinitargs

Bases: object

An old-style class with a __getinitargs__ method. Used for testing explain_pickle.

class sage.misc.explain_pickle.TestReduceNoGetinitargs

Bases: object

An old-style class with no __getinitargs__ method. Used for testing explain_pickle.

```python
sage.misc.explain_pickle.explain_pickle(pickle=None, file=None, compress=True, **kwargs)
```

Explain a pickle. That is, produce source code such that evaluating the code is equivalent to loading the pickle. Feeding the result of `explain_pickle` to `sage_eval` should be totally equivalent to loading the pickle with cPickle.

INPUT:

- pickle – the pickle to explain, as a string (default: None)
- file – a filename of a pickle (default: None)
- compress – if False, don’t attempt to decompress the pickle (default: True)
- in_current_sage – if True, produce potentially simpler code that is tied to the current version of Sage. (default: False)
- default_assumptions – if True, produce potentially simpler code that assumes that generic unpickling code will be used. This code may not actually work. (default: False)
- eval – if True, then evaluate the resulting code and return the evaluated result. (default: False)
- preparse – if True, then produce code to be evaluated with Sage’s preparsers; if False, then produce standard Python code; if None, then produce code that will work either with or without the preparsers. (default: True)
- pedantic – if True, then carefully ensures that the result has at least as much sharing as the result of cPickle (it may have more, for immutable objects). (default: False)

Exactly one of pickle (a string containing a pickle) or file (the filename of a pickle) must be provided.

EXAMPLES:

```python
sage: explain_pickle(dumps({('a', 'b'): [1r, 2r]}))
{(‘a’, ‘b’):[1r, 2r]}
sage: explain_pickle(dumps(RR(pi)), in_current_sage=True)
from sage.rings.real_mpfr import __create__RealNumber_version0
from sage.rings.real_mpfr import __create__RealField_version0
__create__RealNumber_version0(__create__RealField_version0(53r, False, ‘RNDN’), ’3.
˓→4gvml245kc0@0’, 32r)
sage: s = ’hi’
sage: explain_pickle(dumps((s, s)))
(‘hi’, ‘hi’)
```

(continues on next page)
sage: explain_pickle(dumps((s, s)), pedantic=True)
si = 'hi'
(si, si)
sage: explain_pickle(dumps(5r))
5r
sage: explain_pickle(dumps(5r), preparse=False)
5
sage: explain_pickle(dumps(5r), preparse=None)
int(5)
sage: explain_pickle(dumps(22/7))
pg_make_rational = unpickle_global('sage.rings.rational', 'make_rational')
pg_make_rational('m/7')
sage: explain_pickle(dumps(22/7), in_current_sage=True)
from sage.rings.rational import make_rational
make_rational('m/7')
sage: explain_pickle(dumps(22/7), default_assumptions=True)
from sage.rings.rational import make_rational
make_rational('m/7')

sage.misc.explain_pickle.explain_pickle_string(pickle, in_current_sage=False,
default_assumptions=False, eval=False,
preparse=True, pedantic=False)

This is a helper function for explain_pickle. It takes a decompressed pickle string as input; other than that, its options are all the same as explain_pickle.

EXAMPLES:

sage: sage.misc.explain_pickle.explain_pickle_string(dumps("Hello, world",˓→compress=False))
'Hello, world'

(See the documentation for explain_pickle for many more examples.)

sage.misc.explain_pickle.name_is_valid(name)

Test whether a string is a valid Python identifier. (We use a conservative test, that only allows ASCII identifiers.)

EXAMPLES:

sage: from sage.misc.explain_pickle import name_is_valid
sage: name_is_valid('fred')
True
sage: name_is_valid('Yes!ValidName')
False
sage: name_is_valid('_happy_1234')
True

sage.misc.explain_pickle.test_pickle(p, verbose_eval=False, pedantic=False, args=())

Tests explain_pickle on a given pickle p. p can be:

• a string containing an uncompressed pickle (which will always end with a ‘.’)
• a string containing a pickle fragment (not ending with ‘.’) test_pickle will synthesize a pickle that will push args onto the stack (using persistent IDs), run the pickle fragment, and then STOP (if the string ‘mark’ occurs in args, then a mark will be pushed)
• an arbitrary object; test_pickle will pickle the object
Once it has a pickle, test_pickle will print the pickle’s disassembly, run explain_pickle with in_current_sage=True and False, print the results, evaluate the results, unpickle the object with cPickle, and compare all three results.

If verbose_eval is True, then test_pickle will print messages before evaluating the pickles; this is to allow for tests where the unpickling prints messages (to verify that the same operations occur in all cases).

EXAMPLES:

```python
def test_pickle(l):
    # py2
    protocol = int(code[0][8:], 16)
    # py3
    protocol = int(code[0][0], 16)
    highest_protocol = max([sage.misc.explain_pickle.highest_protocol_for_opcode(o) for o in code])
    current_sage = sage.misc.explain_pickle.current_sage()
    result = [sage.misc.explain_pickle.unpickle_for_opcode(o, current_sage) for o in code]
```

sage.misc.explain_pickle.unpickle_appends(lst, vals)
Given a list (or list-like object) and a sequence of values, appends the values to the end of the list. This is careful to do so using the exact same technique that cPickle would use. Used by explain_pickle.

EXAMPLES:

```python
v = []
u = []
u = []
u = []
```

sage.misc.explain_pickle.unpickle_build(obj, state)
Set the state of an object. Used by explainpickle.

EXAMPLES:

```python
v = EmptyNewstyleClass()
v = EmptyNewstyleClass()
v = EmptyNewstyleClass()
v = EmptyNewstyleClass()
```

sage.misc.explain_pickle.unpickle_extension(code)
Takes an integer index and returns the extension object with that index. Used by explain_pickle.

EXAMPLES:

```python
add_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 42)
```

sage.misc.explain_pickle.unpickle_instantiate(fn, args)
Instantiates a new object of class fn with arguments args. Almost always equivalent to fn(*args). Used by explain_pickle.
EXAMPLES:

```python
sage: unpickle_instantiate(Integer, ('42',))
42
```

```
sage: unpickle_newobj(klass, args)
Create a new object; this corresponds to the C code klass->tp_new(klass, args, NULL). Used by explain_pickle.
EXAMPLES:

sage: unpickle_newobj(tuple, ([1, 2, 3],))
(1, 2, 3)
```

```
sage: unpickle_persistent(s)
Takes an integer index and returns the persistent object with that index; works by calling whatever callable is stored in unpickle_persistent_loader. Used by explain_pickle.
EXAMPLES:

sage: import sage.misc.explain_pickle
sage: sage.misc.explain_pickle.unpickle_persistent_loader = lambda n: n+7
sage: unpickle_persistent(35)
42
```

### 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):

```
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.nested_class_test`.

**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.A3 at ...>
<class 'sage.misc.nested_class.A1.A2.A3'>

sage: nested_pickle(A1)  # py2
<class sage.misc.nested_class.A1 at ...>
sage: nested_pickle(A1)  # py3
<class 'sage.misc.nested_class.A1'>

sage: A1.A2  # py2
<class sage.misc.nested_class.A1.A2 at ...>
sage: A1.A2  # py3
<class 'sage.misc.nested_class.A1.A2'>

<class sage.misc.nested_class.A1.A2.A3 at ...>
<class 'sage.misc.nested_class.A1.A2.A3'>
sage: A1.A2.A3.__name__
'A1.A2.A3'

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)  # py2
<class __main__.A1 at ...>
sage: nested_pickle(B1)  # py2
<class __main__.B1 at ...>
sage: A1.A2  # py2
<class __main__.A1.A2 at ...>
sage: B1.A2  # py2
<class __main__.A1.A2 at ...>
```

(continues on next page)
The name for "A1.A2" could potentially be set to "B1.A2". But that will work anyway.

```
sage: nested_pickle(A1)      # py3
<class '__main__.A1'>
sage: nested_pickle(B1)      # py3
<class '__main__.B1'>
sage: A1.A2                   # py3
<class '__main__.A1.A2'>
sage: B1.A2                   # py3
<class '__main__.A1.A2'>
```

```
5.5. Fixing pickle for nested classes
```

```
sage: from sage.misc.nested_class import *
sage: class A(object):
    ....:     class B(object):
    ....:         pass
sage: module = sys.modules['__main__']
sage: A.B.__name__
'B'
sage: getattr(module, 'A.B', 'Not found')
'Not found'
sage: modify_for_nested_pickle(A, 'A', module)
```

```
sage: A.B.__name__
'A.B'
sage: getattr(module, 'A.B', 'Not found')
<class '__main__.A.B'>
```

Here we demonstrate the effect of the `first_run` argument:

```
sage: modify_for_nested_pickle(A, 'X', 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'
```

Note that the class is now found in the module under both its old and its new name:
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
import nested_pickle
module = sys.modules['__main__']
class A(object):
    class B:
        pass
nested_pickle(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
A.B.__name__
getattr(module, 'A.B', 'Not found')  # py2
<class '__main__.A.B at ...'>
getattr(module, 'A.B', 'Not found')  # py3
<class '__main__.A.B'>
```

In Python 2.6, decorators work with classes; then `@nested_pickle` should work as a decorator:

```python
@nested_pickle
class A2(object):
    class B:
        pass
A2.B.__name__  # todo: not implemented
getattr(module, 'A2.B', 'Not found')  # todo: not implemented
<class '__main__.A2.B at ...'>
```

EXAMPLES:

```python
from sage.misc.nested_class import *
loads(dumps(MainClass.NestedClass()))  # indirect doctest
<sage.misc.nested_class.MainClass.NestedClass object at 0x...>
```

A metaclass for nested pickling.

Check that one can use a metaclass to ensure nested_pickle is called on any derived subclass:
```python
sage: from sage.misc.nested_class import NestedClassMetaclass
sage: class ASuperClass(object):  # py2
....:     __metaclass__ = NestedClassMetaclass  # py2
sage: class ASuperClass(object, metaclass=NestedClassMetaclass):  # py3
....:     pass  # py3
sage: class A3(ASuperClass):
....:     class B(object):
....:         pass
sage: A3.B.__name__  # py2
'A3.B'
sage: getattr(sys.modules['__main__'], 'A3.B', 'Not found')  # py2
Not found
sage: A3.B.__name__  # py3
'__main__.A3.B'
```

```python
class sage.misc.nested_class.MainClass
Bases: object

A simple class to test nested_pickle.

EXAMPLES:
```
```python
sage: from sage.misc.nested_class import *
sage: loads(dumps(MainClass()))  # py2
<sage.misc.nested_class.MainClass object at 0x...>
```
```python
class NestedClass
Bases: object

EXAMPLES:
```
```python
sage: from sage.misc.nested_class import *
sage: loads(dumps(MainClass().NestedClass()))  # py2
<sage.misc.nested_class.MainClass.NestedClass object at 0x...>
```
```python
class NestedSubClass
Bases: object

EXAMPLES:
```
```python
sage: from sage.misc.nested_class import *
sage: loads(dumps(MainClass().NestedClass().NestedSubClass()))  # py2
<sage.misc.nested_class.MainClass.NestedClass.NestedSubClass object at 0x...>
```
```python
dummy(x, r=(1, 2, 3.4), *args, **kwds)
A dummy method to demonstrate the embedding of method signature for nested classes.
```

5.5. Fixing pickle for nested classes 343
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:

```
sage: reset()
sage: w = 2/3; w
2/3
sage: show_identifiers()
['w']
```

We next save this session. We are using a file in SAGE_TMP. We do this for testing only — please do not do this, when you want to save your session permanently, since SAGE_TMP will be removed when leaving Sage!

```
sage: save_session(os.path.join(SAGE_TMP, 'session'))
```

This saves a dictionary with w as one of the keys:

```
sage: z = load(os.path.join(SAGE_TMP, 'session'))
sage: list(z)
['w']
sage: z['w']
2/3
```

Next we reset the session, verify this, and load the session back:

```
sage: reset()
sage: show_identifiers()
[]
sage: load_session(os.path.join(SAGE_TMP, 'session'))
```

Indeed w is now defined again:

```
sage: show_identifiers()
['w']
sage: w
2/3
```

It is not needed to clean up the file created in the above code, since it resides in the directory SAGE_TMP.

AUTHOR:

• William Stein

```
sage.misc.session.init(state=None)

Initialize some dictionaries needed by the show_identifiers(), save_session(), and load_session() functions.

INPUT:

• state — a dictionary or None; if None the locals() of the caller is used.

EXAMPLES:
```

(continues on next page)
When we call `init()` below it reinitializes the internal table, so the `w` we just defined doesn’t count as a new identifier:

```
sage: sage.misc.session.init()
sage: show_identifiers()
[]
```

```
sage.misc.session.load_session(name='sage_session', verbose=False)
```

Load a saved session.

This merges in all variables from a previously saved session. It does not clear out the variables in the current sessions, unless they are overwritten. You can thus merge multiple sessions, and don’t necessarily loose all your current work when you use this command.

**Note:** In the Sage notebook the session name is searched for both in the current working cell and the DATA directory.

**EXAMPLES:**

```
sage: a = 5
sage: f = lambda x: x^2
```

For testing, we use a temporary file, that will be removed as soon as Sage is left. Of course, for permanently saving your session, you should choose a permanent file.

```
sage: tmp_f = tmp_filename()
sage: save_session(tmp_f)
sage: del a; del f
sage: load_session(tmp_f)
sage: print(a)
5
```

Note that `f` does not come back, since it is a function, hence couldn’t be saved:

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

```
sage.misc.session.save_session(name='sage_session', verbose=False)
```

Save all variables that can be saved to the given filename. The variables will be saved to a dictionary, which can be loaded using `load(name)` or `load_session()`.

**Note:**

1. Function and anything else that can’t be pickled is not saved. This failure is silent unless you set `verbose=True`.
2. In the Sage notebook the session is saved both to the current working cell and to the DATA directory.
3. One can still make sessions that can’t be reloaded. E.g., define a class with:
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.

```plaintext
sage: a = 5
sage: tmp_f = tmp_filename()
sage: save_session(tmp_f)
sage: del a
sage: load_session(tmp_f)
sage: print(a)
5
```

We illustrate what happens when one of the variables is a function:

```plaintext
sage: f = lambda x : x^2
sage: save_session(tmp_f)
sage: save_session(tmp_f, verbose=True)
Saving...
Not saving f: f is a function, method, class or type ...
```

Something similar happens for cython-defined functions:

```plaintext
sage: g = cython_lambda('double x', 'x*x + 1.5')
sage: save_session(tmp_f, verbose=True)
Saving...
Not saving g: g is a function, method, class or type ...
```

```
 sage.misc.session.show_identifiers(hidden=False)
Returns a list of all variable names that have been defined during this session. By default, this returns only those identifiers that don’t start with an underscore.

**INPUT:**

- `hidden` – bool (Default: False); If True, also return identifiers that start with an underscore.

Chapter 5. Saving and Loading Sage Objects
A list of variable names

EXAMPLES:
We reset the state of all variables, and see that none are defined:

```sage
sage: reset()
sage: show_identifiers()
[]
```

We then define two variables, one which overwrites the default factor function; both are shown by `show_identifiers()`:

```sage
sage: a = 10
sage: factor = 20
sage: show_identifiers()
['a', 'factor']
```

To get the actual value of a variable from the list, use the `globals()` function:

```sage
sage: globals() ['factor']
20
```

By default `show_identifiers()` only returns variables that don’t start with an underscore. There is an option `hidden` that allows one to list those as well:

```sage
sage: _hello = 10
sage: show_identifiers()
['a', 'factor']
sage: '_hello' in show_identifiers(hidden=True)
True
```

Many of the hidden variables are part of the IPython command history, at least in command line mode:

```sage
sage: show_identifiers(hidden=True)  # random output
['__', '_i', '_6', '_4', '_3', '_l', '_ii', '__doc__', '__builtins__', '__', '_9', 
  '__name__', '_', 'a', '_i12', '_i14', 'factor', '__file__', '_hello', '_i13', '_
  ->i11', '_i10', '_i15', '_i5', '_i3', '_10', '_iii', '_19', '_i8', '_17', '_16', '_
  ->i4', '_i3', '_i2', '_i1', '_init_cmdline', '_14']
```
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:

```python
sage: import sage.misc.banner; sage.misc.banner.SAGE_BANNER = ''
sage: 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 (optional, 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:

```python
sage: print(sage.misc.banner.banner_text(full=True))
+——————————————————————–+
| SageMath version ... |
sage: 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 (optional, default = 0)
• tiny – float (optional, default = 0)
• prerelease – boolean (optional, default = False)
• print_message – boolean (optional, 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
```

sage.misc.banner.version()
Return the version of Sage.

OUTPUT:
str

EXAMPLES:

```
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:
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’: 1.2, ‘prerelease’: False}. If the Sage version is ‘3.2.alpha0’, then the dictionary is {'major': 3, ‘minor’: 2, ‘tiny’: 0, ‘prerelease’: True}.

EXAMPLES:

```
sage: from sage.misc.banner import version_dict
sage: print("SageMath major version is %s" % version_dict()['major'])
SageMath major version is ...
sage: version_dict()['major'] == int(sage.version.version.split('.')[0])
True
```

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:

```
sage: x = 5
sage: reset()
sage: x
x
sage: fn = tmp_filename(ext='foo.py')
sage: sage.misc.reset.EXCLUDE.add('fn')
sage: with open(fn, 'w') as f:
    ....:_ = f.write('a = 111')
sage: attach(fn)
sage: [fn] == attached_files()
True
```
sage: reset()
sage: [fn] == attached_files()
True
sage: reset(attached=True)
sage: [fn] == attached_files()
False
sage: 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')
sage: restore('x y')
sage: x
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
(x, Rational Field, 15)
sage: restore('ww')
sage: ww
Traceback (most recent call last):
...  
NameError: name 'ww' is not defined
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
```

On OS X, PDFs are opened by default using the ‘open’ command, which runs whatever has been designated as the PDF viewer in the OS. To change this to use ‘Adobe Reader’:

```
sage: viewer.pdf_viewer('open -a /Applications/Adobe\ Reader.app') # not tested
```

Similarly, you can set `viewer.browser(...)`, `viewer.dvi_viewer(...)`, and `viewer.png_viewer(...)`. You can make this change permanent by adding lines like these to your `SAGE_STARTUP_FILE` (which is `$HOME/.sage/init.sage` by default):

```
from sage.misc.viewer import viewer
viewer.pdf_viewer('open -a /Applications/Adobe\ Reader.app')
```

### Functions and classes

#### class sage.misc.viewer.Viewer

Bases: `sage.structure.sage_object.SageObject`

Set defaults for various viewing applications: a web browser, a dvi viewer, a pdf viewer, and a png viewer.

**Examples:**

```
sage: from sage.misc.viewer import viewer
sage: old_browser = viewer.browser()  # indirect doctest
sage: viewer.browser('open -a /Applications/Firefox.app')
sage: viewer.browser()  
'open -a /Applications/Firefox.app'
sage: viewer.browser(old_browser)  # restore old value
```

#### browser(app=None)

Change the default browser. Return the current setting if arg is `None`, which is the default.

**Input:**

- app – `None` or a string, the program to use

**Examples:**

```
sage: from sage.misc.viewer import viewer
sage: old_browser = viewer.browser()  
'open -a /Applications/Firefox.app'
sage: viewer.browser(old_browser)  # restore old value
```

#### dvi_viewer(app=None)

Change the default dvi viewer. Return the current setting if arg is `None`, which is the default.

**Input:**

- app – `None` or a string, the program to use

6.1. Interactive Sage Sessions
EXAMPLES:

```python
sage: from sage.misc.viewer import viewer
sage: old_dvi_app = viewer.dvi_viewer()
sage: viewer.dvi_viewer('/usr/bin/xdvi') # indirect doctest
sage: viewer.dvi_viewer('/usr/bin/xdvi')
sage: viewer.dvi_viewer(old_dvi_app) # restore old value
```

**pdf_viewer(app=None)**

Change the default pdf viewer. Return the current setting if `app` is None, which is the default.

**INPUT:**

- `app` – None or a string, the program to use

**EXAMPLES:**

```python
sage: from sage.misc.viewer import viewer
sage: old_pdf_app = viewer.pdf_viewer()
sage: viewer.pdf_viewer('/usr/bin/pdfopen') # indirect doctest
sage: viewer.pdf_viewer('/usr/bin/pdfopen')
sage: viewer.pdf_viewer(old_pdf_app) # restore old value
```

**png_viewer(app=None)**

Change the default png viewer. Return the current setting if `app` is None, which is the default.

**INPUT:**

- `app` – None or a string, the program to use

**EXAMPLES:**

```python
sage: from sage.misc.viewer import viewer
sage: old_png_app = viewer.png_viewer()
sage: viewer.png_viewer('display') # indirect doctest
sage: viewer.png_viewer('display')
sage: viewer.png_viewer(old_png_app) # restore old value
```

**sage.misc.viewer.browser()**

Return the program used to open a web page. By default, the program used depends on the platform and other factors, like settings of certain environment variables. To use a different program, call `viewer.browser('PROG')`, where ‘PROG’ is the desired program.

This will start with ‘sage-native-execute’, which sets the environment appropriately.

**EXAMPLES:**

```python
sage: from sage.misc.viewer import browser
sage: browser() # random -- depends on OS, etc.
'sage-native-execute sage-open'
sage: browser().startswith('sage-native-execute')
True
```

**sage.misc.viewer.default_viewer(viewer=None)**

Set up default programs for opening web pages, PDFs, PNGs, and DVI files.
INPUT:

- viewer: None or a string: one of ‘browser’, ‘pdf’, ‘png’, ‘dvi’ – return the name of the corresponding program. None is treated the same as ‘browser’.

EXAMPLES:

```python
sage: from sage.misc.viewer import default_viewer
sage: default_viewer(None) # random -- depends on OS, etc.
'sage-open'
sage: default_viewer('pdf') # random -- depends on OS, etc.
'xdg-open'
sage: default_viewer('jpg')
Traceback (most recent call last):
...
ValueError: Unknown type of viewer: jpg.
```

`sage.misc.viewer.dvi_viewer()`

Return the program used to display a dvi file. By default, the program used depends on the platform and other factors, like settings of certain environment variables. To use a different program, call viewer.
dvi_viewer('PROG'), where ‘PROG’ is the desired program.

This will start with ‘sage-native-execute’, which sets the environment appropriately.

EXAMPLES:

```python
sage: from sage.misc.viewer import dvi_viewer
sage: dvi_viewer() # random -- depends on OS, etc.
'sage-native-execute sage-open'
sage: dvi_viewer().startswith('sage-native-execute')
True
```

`sage.misc.viewer.pdf_viewer()`

Return the program used to display a pdf file. By default, the program used depends on the platform and other factors, like settings of certain environment variables. To use a different program, call viewer.
pdf_viewer('PROG'), where ‘PROG’ is the desired program.

This will start with ‘sage-native-execute’, which sets the environment appropriately.

EXAMPLES:

```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()
'sage-native-execute acroread'
sage: viewer.pdf_viewer('old_pdf_app')
```

`sage.misc.viewer.png_viewer()`

Return the program used to display a png file. By default, the program used depends on the platform and other factors, like settings of certain environment variables. To use a different program, call viewer.
png_viewer('PROG'), where ‘PROG’ is the desired program.

This will start with ‘sage-native-execute’, which sets the environment appropriately.

EXAMPLES:
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.

\[ \text{sage.misc.pager.pager()} \]

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.

\[ \text{sage.misc.sagedoc.detex(s, embedded=False)} \]
This strips LaTeX commands from a string; it is used by the format function to process docstrings for display from the command line interface.

INPUT:
- s - string
- embedded - boolean (optional, default False)

If embedded is False, then do the replacements in both math_substitutes and nonmath_substitutes. If True, then only do nonmath_substitutes.

OUTPUT:
string

EXAMPLES:

\[ \text{sage: from sage.misc.sagedoc import detex} \]
\[ \text{sage: detex(\texttt{r'\textbf{Some math: } x \mapsto y \text{. \{\textbf{Bold face}\}.}')} \]
'Some math: x |--> y. { Bold face}.}'

\[ \text{sage: detex(\texttt{r'\textbf{Some math: } n \geq k \text{. A website: } \url{sagemath.org}.}') \]
'Some math: n >= k. A website: sagemath.org.'
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(\lvert x\ast y \rvert\right)`')
'(| x * y |)'
sage: detex(r'`\left(\leq\le\leftarrow \rightarrow\unknownmacro\to\right)`')
'(<=<=<-- -->\unknownmacro-->)

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 (optional, default False)

OUTPUT: string

Set embedded equal to True if formatting for use in the notebook; this just gets passed as an argument to detex.

See also:

sage.misc.sageinspect.sage_getdoc() to get the formatted documentation of a given object.

EXAMPLES:

sage: from sage.misc.sagedoc import format
sage: identity_matrix(2).rook_vector.__doc__[202:274]
'Let `A` be an `m` by `n` (0,1)-matrix. We identify `A` with a chessboard'
sage: format(identity_matrix(2).rook_vector.__doc__[202:274])
'Let A be an m by n (0,1)-matrix. We identify A with a chessboard'

If the first line of the string is `nodetex`, remove `nodetex` but don’t modify any TeX commands:

sage: format("nodetex\n\x \\geq \y")
'\x \ \geq \y'

Testing a string enclosed in triple angle brackets:

sage: format("<<<identity_matrix)>>")
'<<<identity_matrix\n'
sage: format("identity_matrix>>>")
'identity_matrix>>>\n'
sage: format("<<<identity_matrix>>>")[:28]
'Definition: identity_matrix('
sage.misc.sagedoc.format_search_as_html(what, results, search)

Format the output from search_src, search_def, or search_doc as html, for use in the notebook.

INPUT:

• what - (string) what was searched (source code or documentation)
• results - (string or list) the results of the search as a string or list of search results
• search - (string or list) what was being searched for, either as a string which is taken verbatim, or a list of multiple search terms if there were more than one

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:

```python
code: from sage.misc.sagedoc import format_search_as_html
code: format_search_as_html('Source', 'algebras/steenrod_algebra_element.py:
  → an antihomomorphism: if we call the antipode `c`, then', 'antipode
  → antihomomorphism')
  
  '<html><font color="black"><h2>Search Source: "antipode antihomomorphism"
  → antihomomorphism"
  </h2></font>
  <font color="darkpurple"><ol><li><a href="/src/algebras/steenrod_algebra_element.py" target="_blank">algebras/steenrod_algebra_element.py</a></li>
  </ol></font></html>

code: 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')
  
  '<html><font color="black"><h2>Search Other: "antipode antihomomorphism"
  → antihomomorphism"
  </h2></font>
  </ol></font></html>
```

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
code: from sage.misc.sagedoc import format_src
code: format_src('unladen swallow')
  
  'unladen swallow'

code: format_src('<<<Sq>>>')[:5:15]
  
  'Sq(*nums):'
```

sage.misc.sagedoc.help(module=None)

If there is an argument module, print the Python help message for module. With no argument, print a help message about getting help in Sage.

EXAMPLES:

```python
code: 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:

```
sage: from sage.misc.sagedoc import my_getsource
sage: s = my_getsource(identity_matrix)
sage: s[15:34]
'def identity_matrix'
```

sage.misc.sagedoc.process_dollars(s)
Replace dollar signs with backticks.

More precisely, do a regular expression search. Replace a plain dollar sign ($) by a backtick ('). Replace an escaped dollar sign (\$) by a dollar sign ($). Don’t change a dollar sign preceded or followed by a backtick ('\$' or '$'), because of strings like “$HOME”. Don’t make any changes on lines starting with more spaces than the first nonempty line in s, because those are indented and hence part of a block of code or examples.

This also doesn’t replaces dollar signs enclosed in curly braces, to avoid nested math environments.

EXAMPLES:

```
sage: from sage.misc.sagedoc import process_dollars
sage: process_dollars('hello')
'hello'
sage: process_dollars('some math: $x=y$')
'some math: `x=y`'
```

Replace $ with $, and don’t do anything when backticks are involved:

```
sage: process_dollars(r'a `\$REAL\$` dollar sign: \$')
'a `\$REAL\$` dollar sign: '$
```

Don’t make any changes on lines indented more than the first nonempty line:

```
sage: s = '\n first line\n indented $x=y$'
sage: s == process_dollars(s)
True
```

Don’t replace dollar signs enclosed in curly braces:

```
sage: process_dollars(r'f(n) = 0 \text{ if $n$ is prime}')
'f(n) = 0 \text{ if $n$ is prime}'
```

This is not perfect:

```
sage: process_dollars(r'f(n) = 0 \text{ if $n$ is prime}$')
'f(n) = 0 \text{ if $n$ is prime}$'
```
The regular expression search doesn’t find the last $. Fortunately, there don’t seem to be any instances of this kind of expression in the Sage library, as of this writing.

`sage.misc.sagedoc.process_extlinks(s, embedded=False)`

In docstrings at the command line, process markup related to the Sphinx extlinks extension. For example, replace :trac:`NUM` with https://trac.sagemath.org/NUM, and similarly with :python:`TEXT` and :wikipedia:`TEXT`, looking up the url from the dictionary `extlinks` in `sage.docs.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 (optional, default False)

This function is called by `format()`, and if in the notebook, it sets `embedded` to be True, otherwise False.

**EXAMPLES:**

```python
sage: from sage.misc.sagedoc import process_extlinks
'See https://trac.sagemath.org/1234, https://en.wikipedia.org/wiki/Sage_→(mathematics_software), and https://trac.sagemath.org/4321 ...

sage: process_extlinks('See :trac:`1234` for more information.', embedded=True)
'See :trac:`1234` for more information.'

sage: process_extlinks('see :python:`Implementing Descriptors <reference/datamodel.html#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

This function is called by `format()`

**EXAMPLES:**

```python
sage: from sage.misc.sagedoc import process_mathtt
sage: process_mathtt(r'\texttt{e^self}')
'e^self'
```

`sage.misc.sagedoc.search_def(name, extra1='', extra2='', extra3='', extra4='', extra5='', **kwds)`

Search Sage library source code for function definitions containing `name`. The search is case-insensitive by default.

**INPUT:** same as for `search_src()`.

**OUTPUT:** same as for `search_src()`.

**Note:** The regular expression used by this function only finds function definitions that are preceded by spaces, so if you use tabs on a “def” line, this function will not find it. As tabs are not allowed in Sage library code, this should not be a problem.
EXAMPLES:

See the documentation for \texttt{search\_src()} for more examples.

\begin{verbatim}
sage: print(search\_def("fetch", interact=False)) # random # long time
matrix/matrix0.pyx: cdef fetch(self, key):
matrix/matrix0.pxd: cdef fetch(self, key):
sage: print(search\_def("fetch", path\_re="pyx", interact=False)) # random # long time
matrix/matrix0.pyx: cdef fetch(self, key):
\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 \texttt{string}. The search is case-insensitive by default.

The file paths in the output are relative to \$SAGE\_DOC.

INPUT: same as for \texttt{search\_src()}.

OUTPUT: same as for \texttt{search\_src()}.

EXAMPLES:

See the documentation for \texttt{search\_src()} for more examples.

\begin{verbatim}
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...
</p>Sage to use (the string)
\end{verbatim}

If you search the documentation for \texttt{tree}, then you will get too many results, because many lines in the documentation contain the word \texttt{toctree}. If you use the \texttt{whole\_word} option, though, you can search for \texttt{tree} without returning all of the instances of \texttt{toctree}. In the following, since \texttt{search\_doc('tree', interact=False)} returns a string with one line for each match, counting the length of \texttt{search\_doc('tree', interact=False).splitlines()} gives the number of matches.

\begin{verbatim}
sage: N = len(search\_doc('tree', interact=False).splitlines()) # optional ~
dochtml, long time
sage: L = search\_doc('tree', whole\_word=True, interact=False).splitlines() # optional ~
dochtml, long time
sage: len(L) < N # optional ~
dochtml, long time
True
sage: import re
sage: tree\_re = re.compile(r'(^|\W)tree(\W|$)', re.I)
sage: all(tree\_re.search(l) for l in L) # optional ~
dochtml, long time
True
\end{verbatim}

\begin{verbatim}
sage.misc.sagedoc.search\_src(string, extra1="", extra2="", extra3="", extra4="", extra5="", **kwds)
\end{verbatim}

Search Sage library source code for lines containing \texttt{string}. The search is case-insensitive by default.

INPUT:

- \texttt{string} - a string to find in the Sage source code.
- \texttt{extra1},..., \texttt{extra5} - additional strings to require when searching. Lines must match all of these, as well as \texttt{string}.
- \texttt{whole\_word} (optional, default False) - if True, search for \texttt{string} and \texttt{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.

6.1. Interactive Sage Sessions
• `ignore_case` (optional, default True) - if False, perform a case-sensitive search

• `multiline` (optional, default False) - if True, search more than one line at a time. In this case, print any matching file names, but don’t print line numbers.

• `interact` (optional, default True) - if False, return a string with all the matches. Otherwise, this function returns None, and the results are displayed appropriately, according to whether you are using the notebook or the command-line interface. You should not ordinarily need to use this.

• `path_re` (optional, default '') - regular expression which the filename (including the path) must match.

• `module` (optional, default ‘sage’) - the module in which to search. The default is ‘sage’, the entire Sage library. If `module` doesn’t start with “sage”, then the links in the notebook output may not function.

OUTPUT: If `interact` is False, then return a string with all of the matches, separated by newlines. On the other hand, if `interact` is True (the default), there is no output. Instead: at the command line, the search results are printed on the screen in the form `filename:line_number:line of text`, showing the filename in which each match occurs, the line number where it occurs, and the actual matching line. (If `multiline` is True, then only the filename is printed for each match.) The file paths in the output are relative to `$SAGE_SRC`. In the notebook, each match produces a link to the actual file in which it occurs.

The string and `extraN` arguments are treated as regular expressions, as is `path_re`, and errors will be raised if they are invalid. The matches will be case-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.

**EXAMPLES:**
First note that without using `interact=False`, this function produces no output, while with `interact=False`, the output is a string. These examples almost all use this option, so that they have something to which to compare their output.

You can search for “matrix” by typing `search_src("matrix")`. This particular search will produce many results:

```
sage: len(search_src("matrix", interact=False).splitlines()) # random # long time
9522
```

You can restrict to the Sage calculus code with `search_src("matrix", module="sage.calculus")`, and this produces many fewer results:

```
sage: len(search_src("matrix", module="sage.calculus", interact=False).splitlines()) # random
26
```

Note that you can do tab completion on the `module` string. Another way to accomplish a similar search:

```
sage: len(search_src("matrix", path_re="calc", interact=False).splitlines()) > 15
True
```

The following produces an error because the string ‘fetch()’ is a malformed regular expression:

```
sage: print(search_src(" fetch(" "def", interact=False)) # py2
Traceback (most recent call last):
  ...
error: unbalanced parenthesis
```

(continues on next page)
To fix this, escape the parenthesis with a backslash:

```
sage: print(search_src(r" fetch\(", "def", interact=False)) # random # long time
```
```
matrix/matrix0.pyx: cdef fetch(self, key):
```
```
sage: print(search_src(r" fetch\(", "def", "pyx", interact=False)) # random # long time
```
```
matrix/matrix0.pyx: cdef fetch(self, key):
```

As noted above, the search is case-insensitive, but you can make it case-sensitive with the ‘ignore_case’ key word:

```
sage: s = search_src('Matrix', path_re='matrix', interact=False); s.find('x') > 0
True
```
```
sage: s = search_src('MatRiX', path_re='matrix', interact=False); s.find('x') > 0
True
```
```
sage: s = search_src('MatRiX', path_re='matrix', interact=False, ignore_case=False); s.find('x') > 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()) # random
```
```
misc/sagedoc.py:... len(search_src("matrix", module="sage.calculus", interact=False).splitlines()) # random
```
```
misc/sagedoc.py:... len(search_src("matrix", path_re="calc", interact=False).splitlines()) > 15
```
```
misc/sagedoc.py:... print(search_src(" fetch\(", "def", interact=False)) # py2
```
```
misc/sagedoc.py:... print(search_src(" fetch\(", "def", interact=False)) # py3
```
```
misc/sagedoc.py:... print(search_src(r" fetch\(", "def", interact=False)) # random
```
```
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
\n'
sage: test = 'TESTS: \nHere is a test:\n\nsage: 2+2 \n5 \n'
sage: test2 = 'TESTS:: \n\nsage: 2+2 \n6 \n'
```

Test lines starting with “REFERENCES:”:

```python
sage: refs = 'REFERENCES: \n\ntext text \n'
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
```
Test Sphinx directives:

```python
tsage: directive = ' .. todo:: \n do some stuff \n'
tsage: skip_TESTS_block(start + test + refs + test2 + directive).rstrip() == (start + refs + directive).rstrip()
True
```

Test unindented lines:

```python
tsage: unindented = ' NOT INDENTED\n' 
tsage: skip_TESTS_block(start + test + unindented).rstrip() == (start + unindented).rstrip()
True
```

Test headers:

```python
tsage: header = ' Header:\n ~~~~~~~' 
tsage: skip_TESTS_block(start + test + header) == start + header
True
```

Not a header because the characters on the second line must all be the same:

```python
tsage: fake_header = ' Header:\n -*-=-=-=-=' 
tsage: skip_TESTS_block(start + test + fake_header).rstrip() == start.rstrip()
True
```

Not a header because it’s indented compared to ‘TEST’ in the string test:

```python
tsage: another_fake = '\n blah\n ----' 
tsage: skip_TESTS_block(start + test + another_fake).rstrip() == start.rstrip()
True
```

Double colons :: are also not considered as headers (trac ticket #27896):

```python
tsage: colons = '::\n\n\n sage: 2+2\n 4\n\n'
tsage: skip_TESTS_block(start + test2 + colons).rstrip() == start.rstrip()
True
```
6.1.6 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

```python
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 (optional, 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">
    A test</div>'
```

```python
sage: sphinxify('**Testing**

`monospace`
')
'<div class="docstring">
    <strong>Testing</strong>'
```

```python
sage: sphinxify('
x=y`
')
'<div class="docstring">
    x=y
</div>'
```

```python
sage: sphinxify('`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:
```python
sage: from sage.misc.package import list_packages
sage: pkgs = list_packages(local=True)  # optional - build
sage: sorted(pkgs.keys())  # optional - build, random
['4ti2',
 'alabaster',
 'arb',
 ...
 'zlib',
 'zn_poly']
```

### Functions

#### exception

```python
definition
sage.misc.package.PackageNotFoundError(*args)
Bases: RuntimeError
```

This class defines the exception that should be raised when a function, method, or class cannot detect a Sage package that it depends on.

This exception should be raised with a single argument, namely the name of the package.

When a `PackageNotFoundError` is raised, this means one of the following:
- The required optional package is not installed.
- The required optional package is installed, but the relevant interface to that package is unable to detect the package.

Raising a `PackageNotFoundError` is deprecated. Use `sage.features.FeatureNotPresentError` instead.

User code can continue to catch `PackageNotFoundError` exceptions for compatibility with older versions of the Sage library. This does not cause deprecation warnings.

**EXAMPLES:**

```python
sage: from sage.misc.package import PackageNotFoundError
sage: try:
    ...:     pass
...: except PackageNotFoundError:
    ...:     pass
```

### sage.misc.package.experimental_packages()

Return two lists. The first contains the installed and the second contains the not-installed experimental packages that are available from the Sage repository.

**OUTPUT:**

- installed experimental packages (as a list)
- NOT installed experimental packages (as a list)

Run `sage -i package_name` from a shell to install a given package or `sage -f package_name` to re-install it.

**See also:**

`sage.misc.package.list_packages()`

**EXAMPLES:**
sage: from sage.misc.package import experimental_packages
sage: installed, not_installed = experimental_packages()  # optional - build
doctest:...: DeprecationWarning: ...

```
sage.misc.package.installed_packages(exclude_pip=True)
Return a dictionary of all installed packages, with version numbers.

INPUT:

• exclude_pip – (optional, default: True) whether “pip” packages are excluded from the list

EXAMPLES:

```
sage: sorted(installed_packages().keys())  # optional - build
[...'alabaster',...'sage_conf',...]
sage: installed_packages()['alabaster']  # optional - build, random
'0.7.12'
```

See also:

`sage.misc.package.list_packages()`

```
sage.misc.package.is_package_installed(package, exclude_pip=True)
Return whether (any version of) package is installed.

INPUT:

• package – the name of the package
• exclude_pip – (optional, default: True) whether to consider pip type packages

EXAMPLES:

```
sage: is_package_installed('gap')  # optional - build
True
```

Giving just the beginning of the package name is not good enough:

```
sage: is_package_installed('matplotli')  # optional - build
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.list_packages(*pkg_types, **opts)
Return a dictionary of information about each package.

The keys are package names and values are dictionaries with the following keys:

• 'type': either 'base', 'standard', 'optional', or 'experimental'
• 'source': either 'normal', '', 'pip', or 'script'
• 'installed': boolean
• 'installed_version': None or a string
```

368 Chapter 6. Interactive Usage Support
• 'remote_version': string

INPUT:
• 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.
• 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.
• local – (optional, default: False) if set to True, then do not consult remote (PyPI) repositories for package versions (only applicable for 'pip' type)
• exclude_pip – (optional, default: False) if set to True, then pip packages are not considered. This is the same as removing 'pip' from pkg_sources.
• ignore_URLError – (default: False) if set to True, then connection errors will be ignored

EXAMPLES:

```python
sage: from sage.misc.package import list_packages
sage: L = list_packages('standard')  # optional - build
sage: sorted(L.keys())  # optional - build, random
['alabaster',
 'arb',
 'babel',
 ...
 'zn_poly']
sage: sage_conf_info = L['sage_conf']  # optional - build
sage: sage_conf_info['type']  # optional - build
'standard'
sage: sage_conf_info['installed']  # optional - build
True
sage: sage_conf_info['source']  # optional - build
'script'
sage: L = list_packages(pkg_sources=['pip'], local=True)  # optional - build,
→internet
sage: bs4_info = L['beautifulsoup4']  # optional - build internet
sage: bs4_info['type']  # optional - build internet
'optional'
sage: bs4_info['source']  # optional - build internet
'pip'
```

Check the option exclude_pip:

```python
sage: [p for p, d in list_packages('optional', exclude_pip=True).items() #
→optional - build
....: if d['source'] == 'pip']
[]
```

`sage.misc.package.optional_packages()`
Return two lists. The first contains the installed and the second contains the not-installed optional packages that are available from the Sage repository.

OUTPUT:
• installed optional packages (as a list)
• NOT installed optional packages (as a list)

Run `sage -i package_name` from a shell to install a given package or `sage -f package_name` to re-install it.

See also:

`sage.misc.package.list_packages()`

EXAMPLES:

```python
sage: from sage.misc.package import optional_packages
sage: installed, not_installed = optional_packages()  # optional - build
doctest::...: DeprecationWarning: ...
sage: 'beautifulsoup4' in installed+not_installed  # optional - build
True
sage: 'beautifulsoup4' in installed  # optional - build beautifulsoup4
True
```

`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: sagetex_manifest = package_manifest('sagetex')  # optional - build
sage: 'files' in sagetex_manifest  # optional - build
True
```

Test a nonexistent package:

```python
sage: package_manifest('dummy-package')  # optional - build
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:

```
sage: std = package_versions('standard', local=True) # optional - build
sage: 'gap' in std # optional - build
True
sage: std['zn_poly'] # optional - build, random
('0.9.p12', '0.9.p12')
```

```
sage.misc.package.pip_installed_packages(normalization=None)
```

Return a dictionary \(name \rightarrow version\) of installed pip packages.

This command returns all pip-installed packages. Not only Sage packages.

INPUT:

- `normalization` – (optional, 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: from sage.misc.package import pip_installed_packages
sage: d = pip_installed_packages() # optional - build
sage: 'scipy' in d # optional - build
True
sage: d['scipy'] # optional - build
'...

sage: d['beautifulsoup4'] # optional - build beautifulsoup4
'...

sage: d['prompt-toolkit'] # optional - build
'...

sage: d = pip_installed_packages(normalization='spkg') # optional - build
sage: d['prompt_toolkit'] # optional - build
'...
```

```
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. trac ticket #19213):

```
sage: from sage.misc.package import pip_remote_version
sage: pip_remote_version('beautifulsoup4') # optional - internet # not tested
u'...
```

These tests are reliable since the tested package does not exist:

6.2. Distribution 371
```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
doctest:...: UserWarning: failed to fetch the version of pkg='hey_this_is_NOT_a_python_package' at http://this.is.not.pypi.com/.../json
```
```
sage: pip_remote_version(nap, pypi_url=pypi, ignore_URLError=False) # optional - internet
Traceback (most recent call last):
... HTTPError: HTTP Error 404: Not Found
```

```python
sage.misc.package.pkgname_split(name)
```
Split a pkgname into a list of strings, ‘name, version’.
For some packages, the version string might be empty.

**EXAMPLES:**

```python
sage: from sage.misc.package import pkgname_split
sage: pkgname_split('hello_world-1.2')
['hello_world', '1.2']
```

```python
sage.misc.package.standard_packages()
```
Return two lists. The first contains the installed and the second contains the not-installed standard packages that are available from the Sage repository.

**OUTPUT:**

- installed standard packages (as a list)
- NOT installed standard packages (as a list)

Run `sage -i package_name` from a shell to install a given package or `sage -f package_name` to re-install it.

**See also:**

```python
sage.misc.package.list_packages()
```

**EXAMPLES:**

```python
sage: from sage.misc.package import standard_packages
sage: installed, not_installed = standard_packages() # optional - build
doctest:....: DeprecationWarning: ...
sage: 'numpy' in installed # optional - build
True
```
6.2.2 Installing shortcut scripts

`sage.misc.dist.install_scripts(directory=None, ignore_existing=False)`

Running `install_scripts(directory)` creates scripts in the given directory that run various software components included with Sage. Each of these scripts essentially just runs `sage --CMD` where `CMD` is also the name of the script:

- ‘gap’ runs GAP
- ‘gp’ runs the PARI/GP interpreter
- ‘hg’ runs Mercurial
- ‘ipython’ runs IPython
- ‘maxima’ runs Maxima
- ‘mwrank’ runs mwrank
- ‘R’ runs R
- ‘singular’ runs Singular
- ‘sqlite3’ runs SQLite version 3
- ‘kash’ runs Kash if it is installed
- ‘M2’ runs Macaulay2 if it is installed

This command:

- verbosely tells you which scripts it adds, and
- will not overwrite any scripts you already have in the given directory.

INPUT:

- `directory` - string; the directory into which to put the scripts. This directory must exist and the user must have write and execute permissions.
- `ignore_existing` - bool (optional, default False): if True, install script even if another version of the program is in your path.

OUTPUT: Verbosely prints what it is doing and creates files in `directory` that are world executable and readable.

**Note:** You may need to run `sage` as root in order to run `install_scripts` successfully, since the user running `sage` needs write permissions on `directory`. Note that one good candidate for `directory` is ‘/usr/local/bin’, so from the shell prompt, you could run

```
sudo sage -c "install_scripts('/usr/local/bin')"
```

**Note:** Running `install_scripts(directory)` will be most helpful if `directory` is in your path.

AUTHORS:

- William Stein: code / design
- Arthur Gaer: design
- John Palmieri: revision, 2011-07 (trac ticket #11602)

EXAMPLES:
6.2.3 Testing for features of the environment at runtime

A computation can require a certain package to be installed in the runtime environment. Abstractly such a package describes a class `Feature` which can be tested for at runtime. It can be of various kinds, most prominently an `Executable` in the PATH or an additional package for some installed system such as a `GapPackage`.

AUTHORS:

- Julian Rüth (2016-04-07): Initial version
- Jeroen Demeyer (2018-02-12): Refactoring and clean up

EXAMPLES:

Some generic features are available for common cases. For example, to test for the existence of a binary, one can use an `Executable` feature:

```python
sage: from sage.features import Executable
sage: Executable(name="sh", executable="sh").is_present()
FeatureTestResult('sh', True)
```

Here we test whether the grape GAP package is available:

```python
sage: from sage.features.gap import GapPackage
sage: GapPackage("grape", spkg="gap_packages").is_present()  # optional: gap_packages
FeatureTestResult('GAP package grape', True)
```

Note that a `FeatureTestResult` acts like a bool in most contexts:

```python
sage: if Executable(name="sh", executable="sh").is_present(): "present."
'present.'
```

When one wants to raise an error if the feature is not available, one can use the `require` method:

```python
sage: Executable(name="sh", executable="sh").require()

sage: Executable(name="random", executable="randomOochoz6x", spkg="random", url="http://...rand.om").require()  # optional - sage_spkg
Traceback (most recent call last):
... try to run...sage -i random...
Further installation instructions might be available at http://rand.om.
```

As can be seen above, features try to produce helpful error messages.

```python
class sage.features.CythonFeature(name, test_code, **kwds)
    Bases: sage.features.Feature

    A Feature which describes the ability to compile and import a particular piece of Cython code.
    To test the presence of name, the cython compiler is run on test_code and the resulting module is imported.
```
EXAMPLES:

```python
sage: from sage.features import CythonFeature
sage: fabs_test_code = ''
    ....: cdef extern from "<math.h>":
    ....:     double fabs(double x)
    ....:
    ....: assert fabs(-1) == 1
    ....: ''
sage: fabs = CythonFeature("fabs", test_code=fabs_test_code, spkg="gcc", url="https://gnu.org")
sage: fabs.is_present()
FeatureTestResult('fabs', True)
```

Test various failures:

```python
sage: broken_code = '''this is not a valid Cython program!''
sage: broken = CythonFeature("broken", test_code=broken_code)
sage: broken.is_present()
FeatureTestResult('broken', False)
```

```python
sage: broken_code = '''cdef extern from "no_such_header_file": pass'''
sage: broken = CythonFeature("broken", test_code=broken_code)
sage: broken.is_present()
FeatureTestResult('broken', False)
```

```python
sage: broken_code = '''import no_such_python_module'''
sage: broken = CythonFeature("broken", test_code=broken_code)
sage: broken.is_present()
FeatureTestResult('broken', False)
```

```python
sage: broken_code = '''raise AssertionError("sorry!")'''
sage: broken = CythonFeature("broken", test_code=broken_code)
sage: broken.is_present()
FeatureTestResult('broken', False)
```

```python
class sage.features.Executable(name, executable, **kwds)
Bases: sage.features.Feature

A feature describing an executable in the PATH.
```

**Note:** Overwrite `is_functional()` if you also want to check whether the executable shows proper behaviour. Calls to `is_present()` are cached. You might want to cache the `Executable` object to prevent unnecessary calls to the executable.

EXAMPLES:

```python
sage: from sage.features import Executable
sage: Executable(name="sh", executable="sh").is_present()
FeatureTestResult('sh', True)
```

`is_functional()`

Return whether an executable in the path is functional.
EXAMPLES:

The function returns True unless explicitly overwritten:

```python
sage: from sage.features import Executable
sage: Executable(name="sh", executable="sh").is_functional()
FeatureTestResult('sh', True)
```

```python
class sage.features.Feature(name, spkg=None, url=None)
Bases: sage.features.TrivialUniqueRepresentation

A feature of the runtime environment

Overwrite _is_present() to add feature checks.

EXAMPLES:

```python
sage: from sage.features.gap import GapPackage
sage: GapPackage("grape", spkg="gap_packages") # indirect doctest
Feature('GAP package grape')
```

For efficiency, features are unique:

```python
sage: GapPackage("grape") is GapPackage("grape")
True
```

```python
is_present()

Return whether the feature is present.

OUTPUT:

A FeatureTestResult which can be used as a boolean and contains additional information about the feature test.

EXAMPLES:

```python
sage: from sage.features.gap import GapPackage
sage: GapPackage("grape", spkg="gap_packages").is_present() # optional: gap_packages
FeatureTestResult('GAP package grape', True)

sage: GapPackage("NOT_A_PACKAGE", spkg="gap_packages").is_present()
FeatureTestResult('GAP package NOT_A_PACKAGE', False)
```

The result is cached:

```python
sage: from sage.features import Feature
sage: class TestFeature(Feature):
....:     def _is_present(self):
....:         print("checking presence")
....:         return True
sage: TestFeature("test").is_present()
checking presence
FeatureTestResult('test', True)

sage: TestFeature("test").is_present()
FeatureTestResult('test', True)

sage: TestFeature("other").is_present()
checking presence
FeatureTestResult('other', True)
```

(continues on next page)
require()
    Raise a :exc:`FeatureNotPresentError` if the feature is not present.

    EXAMPLES:

    sage: from sage.features.gap import GapPackage
    sage: GapPackage("velEeThu").require()
    Traceback (most recent call last):
      ... FeatureNotPresentError: GAP package velEeThu is not available.
      TestPackageAvailability("velEeThu")` evaluated to `fail` in GAP.

resolution()
    Return a suggestion on how to make :meth:`is_present` pass if it did not pass.

    OUTPUT:
    A string, a lazy string, or :const:`None`. The default implementation always returns a lazy string.

    EXAMPLES:

    sage: from sage.features import Executable
    sage: Executable(name="CSDP", spkg="csdp", executable="theta", url="https://github.com/dimpase/csdp").resolution()  # optional - sage_spkg
    l'...To install CSDP...you can try to run...sage -i csdp...Further installation instructions might be available at https://github.com/dimpase/csdp.'

exception sage.features.FeatureNotPresentError
    A missing feature error.

    EXAMPLES:

    sage: from sage.features import Feature, FeatureTestResult
    sage: class Missing(Feature):
    ....:     def _is_present(self):
    ....:         return False
    sage: Missing(name="missing").require()
    Traceback (most recent call last):
      ... FeatureNotPresentError: missing is not available.

class sage.features.FeatureTestResult
    The result of a :meth:`Feature.is_present` call.

    Behaves like a boolean with some extra data which may explain why a feature is not present and how this may be resolved.

    EXAMPLES:
sage: from sage.features.gap import GapPackage
sage: presence = GapPackage("NOT_A_PACKAGE").is_present(); presence # indirect doctest
FeatureTestResult('GAP package NOT_A_PACKAGE', False)
sage: bool(presence)
False

Explanatory messages might be available as reason and resolution:

sage: presence.reason
'`TestPackageAvailability("NOT_A_PACKAGE")` evaluated to `fail` in GAP.'
sage: bool(presence.resolution)
False

If a feature is not present, resolution defaults to feature.resolution() if this is defined. If you do not want to use this default you need explicitly set resolution to a string:

sage: from sage.features import FeatureTestResult
sage: package = GapPackage("NOT_A_PACKAGE", spkg="no_package")
sage: str(FeatureTestResult(package, True).resolution)  # optional - sage_spkg
'...To install GAP package NOT_A_PACKAGE...you can try to run...sage -i no_package...
˓→.
'
sage: str(FeatureTestResult(package, False).resolution)  # optional - sage_spkg
'...To install GAP package NOT_A_PACKAGE...you can try to run...sage -i no_package..
˓→.
'
sage: FeatureTestResult(package, False, resolution="rtm").resolution
'rtm'

class sage.features.PackageSystem(name, spkg=None, url=None)
   Bases: sage.features.Feature

A feature describing a system package manager.

EXAMPLES:

sage: from sage.features import PackageSystem
sage: PackageSystem('conda')
Feature('conda')

spkg_installation_hint(spkgs, prompt, feature)
   Return a string that explains how to install feature.

EXAMPLES:

sage: from sage.features import PackageSystem
sage: homebrew = PackageSystem('homebrew')
sage: homebrew.spkg_installation_hint('openblas')  # optional - SAGE_ROOT
'To install openblas using the homebrew package manager, you can try to run:
˓→brew install openblas'

class sage.features.PipPackageSystem(name, spkg=None, url=None)
   Bases: sage.features.PackageSystem

The feature describing the Pip package manager.

EXAMPLES:
class sage.features.PythonModule(name, **kwds)
Bases: sage.features.Feature
A Feature which describes whether a python module can be imported.

EXAMPLES:
Not all builds of python include the ssl module, so you could check whether it is available:

```python
sage: from sage.features import PythonModule
sage: PythonModule("ssl").require()  # not tested - output depends on the python build
```

class sage.features.SagePackageSystem(name, spkg=None, url=None)
Bases: sage.features.PackageSystem
The feature describing the Sage package manager.

EXAMPLES:

```python
sage: from sage.features import SagePackageSystem
sage: SagePackageSystem()  # optional - sage_spkg
Feature('sage_spkg')
```

class sage.features.StaticFile(name, filename, search_path=None, **kwds)
Bases: sage.features.Feature
A Feature which describes the presence of a certain file such as a database.

EXAMPLES:

```python
sage: from sage.features import StaticFile
sage: StaticFile(name="no_such_file", filename="KaT1aihu", search_path=('/'), spkg="some_spkg", url="http://rand.om").require()  # optional - sage_spkg
Traceback (most recent call last):
... FeatureNotPresentError: no_such_file is not available.
'KaT1aihu' not found in any of ['/']...
To install no_such_file...you can try to run...sage -i some_spkg...
Further installation instructions might be available at http://rand.om.
```

absolute_path()
The absolute path of the file.

EXAMPLES:
A FeatureNotPresentError is raised if the file cannot be found:

```python
sage: from sage.features import StaticFile
sage: StaticFile(name="no_such_file", filename="KaT1aihu", search_path=(), spkg="some_spkg", url="http://rand.om").absolute_path()  # optional - sage_spkg
Traceback (most recent call last):
... FeatureNotPresentError: no_such_file is not available.
'KaT1aihu' not found in any of []...
To install no_such_file...you can try to run...sage -i some_spkg...
Further installation instructions might be available at http://rand.om.
```

**class sage.features.TrivialClasscallMetaClass**

Bases: type

A trivial version of ClasscallMetaClass without Cython dependencies.

**class sage.features.TrivialUniqueRepresentation**

Bases: object

A trivial version of UniqueRepresentation without Cython dependencies.

**sage.features.package_systems()**

Return a list of PackageSystem objects representing the available package systems.

The list is ordered by decreasing preference.

**EXAMPLES:**

```python
sage: from sage.features import package_systems
sage: package_systems()  # random
[Feature('homebrew'), Feature('sage_spkg'), Feature('pip')]
```

### 6.2.4 Checks for bliss

**class sage.features.bliss.Bliss**

Bases: `sage.features.PythonModule`

A Feature which describes whether the `sage.graphs.bliss` module has been enabled for this build of Sage and is functional.

**EXAMPLES:**

```python
sage: from sage.features.bliss import Bliss
sage: Bliss().require()  # optional: bliss
```

**class sage.features.bliss.BlissLibrary**

Bases: `sage.features.CythonFeature`

A Feature which describes whether the Bliss library is present and functional.

**EXAMPLES:**
6.2.5 Testing for CSDP at runtime

```python
sage: from sage.features.csdp import CSDP
sage: CSDP().is_present()  # optional: csdp
FeatureTestResult('CSDP', True)
```

**is_functional()**

Check whether \( \theta \) works on a trivial example.

```python
sage: from sage.features.csdp import CSDP
sage: CSDP().is_functional()  # optional: csdp
FeatureTestResult('CSDP', True)
```

6.2.6 Testing for databases at runtime

```python
sage: from sage.features.databases import DatabaseConwayPolynomials
sage: DatabaseConwayPolynomials().is_present()
```

**is_present()**

Check for the presence of the database.

```python
sage: DatabaseConwayPolynomials().is_present()
FeatureTestResult('Frank Luebeck’s database of Conway polynomials', True)
```

```python
sage: DatabaseCremona().is_present()
# optional: database_cremona_ellcurve
FeatureTestResult('Cremona’s database of elliptic curves', True)
```

6.2. Distribution

381
class sage.features.databases.DatabaseJones
    Bases: sage.features.StaticFile

    A Feature which describes the presence of John Jones’s tables of number fields.

    EXAMPLES:

    sage: from sage.features.databases import DatabaseJones
    sage: bool(DatabaseJones().is_present()) # optional: database_jones_numfield
    True

class sage.features.databases.DatabaseKnotInfo
    Bases: sage.features.PythonModule

    A Feature which describes the presence of the databases at the web-pages KnotInfo and LinkInfo.

    EXAMPLES:

    sage: from sage.features.databases import DatabaseKnotInfo
    sage: DatabaseKnotInfo().is_present() # optional: database_knotinfo
    FeatureTestResult('sage.knots.knotinfo', True)

6.2.7 Checks for FES

class sage.features.fes.LibFES
    Bases: sage.features.PythonModule

    A Feature which describes whether the sage.libs.fes module has been enabled for this build of Sage and is functional.

    EXAMPLES:

    sage: from sage.features.fes import LibFES
    sage: LibFES().require() # optional: fes

class sage.features.fes.LibFESLibrary
    Bases: sage.features.CythonFeature

    A Feature which describes whether the FES library is present and functional.

    EXAMPLES:

    sage: from sage.features.fes import LibFESLibrary
    sage: LibFESLibrary().require() # optional: fes

6.2.8 Check for GAP features

class sage.features.gap.GapPackage
    Bases: sage.features.Feature

    A feature describing the presence of a GAP package.

    EXAMPLES:

    sage: from sage.features.gap import GapPackage
    sage: GapPackage("grape", spkg="gap_packages")
    Feature('GAP package grape')
6.2.9 Check various graph generator programs

class sage.features.graph_generators.Benzene
Bases: sage.features.Executable

A sage.features.graph_generators.Feature which checks for the benzene binary.

EXAMPLES:

```python
sage: from sage.features.graph_generators import Benzene
sage: Benzene().is_present()  # optional: benzene
FeatureTestResult('Benzene', True)
```

is_functional()
Check whether benzene works on trivial input.

EXAMPLES:

```python
sage: from sage.features.graph_generators import Benzene
sage: Benzene().is_functional()  # optional: benzene
FeatureTestResult('Benzene', True)
```

class sage.features.graph_generators.Buckygen
Bases: sage.features.Executable

A sage.features.graph_generators.Feature which checks for the buckygen binary.

EXAMPLES:

```python
sage: from sage.features.graph_generators import Buckygen
sage: Buckygen().is_present()  # optional: buckygen
FeatureTestResult('Buckygen', True)
```

is_functional()
Check whether buckygen works on trivial input.

EXAMPLES:

```python
sage: from sage.features.graph_generators import Buckygen
sage: Buckygen().is_functional()  # optional: buckygen
FeatureTestResult('Buckygen', True)
```

class sage.features.graph_generators.Plantri
Bases: sage.features.Executable

A sage.features.graph_generators.Feature which checks for the plantri binary.

EXAMPLES:

```python
sage: from sage.features.graph_generators import Plantri
sage: Plantri().is_present()  # optional: plantri
FeatureTestResult('plantri', True)
```

is_functional()
Check whether plantri works on trivial input.

EXAMPLES:
6.2.10 Check for lrs

```python
sage: from sage.features.graph_generators import Plantri
sage: Plantri().is_functional()  # optional: plantri
FeatureTestResult('plantri', True)
```

class sage.features.lrs.Lrs

Bases: sage.features.Executable

A `sage.features.Feature` describing the presence of the lrs binary which comes as a part of lrslib.

EXAMPLES:

```python
sage: from sage.features.lrs import Lrs
sage: Lrs().is_present()  # optional: lrslib
FeatureTestResult('lrslib', True)
```

is_functional()

Test whether lrs works on a trivial input.

EXAMPLES:

```python
sage: from sage.features.lrs import Lrs
sage: Lrs().is_functional()  # optional: lrslib
FeatureTestResult('lrslib', True)
```

6.3 Credits

6.3.1 Dependency usage tracking for citations

```python
sage.misc.citation.cython_profile_enabled()
```

Return whether Cython profiling is enabled.

EXAMPLES:

```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
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']
```

6.3.2 License

class sage.misc.copying.License
Bases: object
CHAPTER
SEVEN

DEVELOPMENT TOOLS

7.1 Testing

7.1.1 Unit testing for Sage objects

```python
class sage.misc.sage_unittest.InstanceTester(instance, elements=None, verbose=False, prefix='', max_runs=4096, max_samples=None, **options)
    Bases: unittest.case.TestCase
    A gadget attached to an instance providing it with testing utilities.
    EXAMPLES:
    sage: from sage.misc.sage_unittest import InstanceTester
    sage: InstanceTester(instance = ZZ, verbose = True, elements = [1,2,3])
        Testing utilities for Integer Ring
    This is used by SageObject._tester, which see:
    sage: QQ._tester()
        Testing utilities for Rational Field
    info(message, newline=True)
        Display user information
        EXAMPLES:
        sage: from sage.misc.sage_unittest import InstanceTester
        sage: tester = InstanceTester(ZZ, verbose = True)
        sage: tester.info("hello"); tester.info("world")
        hello world
        sage: tester = InstanceTester(ZZ, verbose = False)
        sage: tester.info("hello"); tester.info("world")
        sage: tester = InstanceTester(ZZ, verbose = True)
        sage: tester.info("hello", newline = False); tester.info(" world")
```

387
runTest()

Trivial implementation of unittest.TestCase.runTest() to please the super class TestCase. That's the price to pay for abusively inheriting from it.

EXAMPLES:

```
sage: from sage.misc.sage_unittest import InstanceTester
sage: tester = InstanceTester(ZZ, verbose = True)
sage: tester.runTest()
```

some_elements(S=None, 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:

```
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)
```

(continues on next page)
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]

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:

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

Test for trac ticket #15919, trac ticket #16244:

sage: Z = IntegerModRing(25)  # random.sample, which was used pre #16244, has a
    threshold at 21!
sage: Z[1]  # since #8389, indexed access is used for ring

Traceback (most recent call last):
...
ValueError: variable name '1' does not start with a letter

sage: tester = InstanceTester(Z, elements=Z, max_runs=5)
sage: list(tester.some_elements())
[0, 1, 2, 3, 4]

sage: C = cartesian_product([Z]*4)
class sage.misc.sage_unittest.PythonObjectWithTests(instance)
Bases: object

Utility class for running basis tests on a plain Python object (that is not in SageObject). More test methods can be added here.

EXAMPLES:

```sage
sage: TestSuite("bla").run()
```

class sage.misc.sage_unittest.TestSuite(instance)
Bases: object

Test suites for Sage objects.

EXAMPLES:

```sage
sage: TestSuite(ZZ).run()
```

No output means that all tests passed. Which tests? In practice this calls all the methods \_\_test\_* of this object, in alphabetic order:

```sage
sage: TestSuite(1).run(\texttt{verbose} = \texttt{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 \texttt{self} is in the category of finite semigroups, this checks that the multiplication is associative (at least on some elements):

```sage
sage: S = FiniteSemigroups().example(alphabet = ('a', 'b'))
sage: TestSuite(S).run(\texttt{verbose} = \texttt{True})
running \_\_test\_an\_\_element() . . . pass
running \_\_test\_associativity() . . . pass
running \_\_test\_cardinality() . . . pass
running \_\_test\_category() . . . pass
running \_\_test\_construction() . . . pass
running \_\_test\_elements() . . .
  Running the test suite of \texttt{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
```

(continues on next page)
The different test methods can be called independently:

```python
sage: S._test_associativity()
```

Debugging tip: in case of failure of some test, use `%pdb on` to turn on automatic debugging on error. Run the failing test 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)
```

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
```

**TODO:**

- Allow for customized behavior in case of failing assertion (warning, error, statistic accounting). This involves reimplementing the methods `fail / failIf / …` of `unittest.TestCase` in `InstanceTester`
- Don’t catch the exceptions if `TestSuite(...) .run()` is called under the debugger, or with `%pdb` on (how to detect this? see `get_ipython()`, `IPython.Magic.shell.call_pdb`, …) In the mean time, see the `catch=False` option.
- Run the tests according to the inheritance order, from most generic to most specific, rather than alphabetically. Then, the first failure will be the most relevant, the others being usually consequences.
- Improve integration with doctests (statistics on failing/passing tests)
- Add proper support for nested testsuites.
- Integration with `unittest`: Make `TestSuite` inherit from `unittest.TestSuite? Make .run(...) accept a result object
• Add some standard option proof = True, asking for the test method to choose appropriately the elements so as to prove the desired property. The test method may assume that a parent implements properly all the super categories. For example, the _test_commutative method of the category CommutativeSemigroups() may just check that the provided generators commute, implicitly assuming that generators indeed generate the semigroup (as required by Semigroups()).

run(category=None, skip=[], catch=True, raise_on_failure=False, **options)
Run all the tests from this test suite:

INPUT:
• category - a category; reserved for future use
• skip - a string or list (or iterable) of strings
• raise_on_failure - a boolean (default: False)
• catch - a boolean (default: True)
All other options are passed down to the individual tests.

EXAMPLES:

```sage```
TestSuite(ZZ).run()
```
We now use the verbose option:

```sage```
TestSuite(1).run(semantic = 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:

```sage```
TestSuite(1).run(semantic = 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
```sage```
TestSuite(1).run(semantic = 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()
```
(continues on next page)
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 ...Blah... failed
------------------------------------------------------------
The following tests failed: _test_b, _test_d, _test_pickling

sage: TestSuite(Blah()).run(verbose = True)
running ._test_a() . . . pass
running ._test_b() . . . fail
Traceback (most recent call last):
...
AssertionError: None
------------------------------------------------------------
running ._test_c() . . . pass
running ._test_category() . . . pass
running ._test_d() . . . fail
Traceback (most recent call last):
...
AssertionError: None
------------------------------------------------------------
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . fail
Traceback (most recent call last):
...
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:

sage: TestSuite(Blah()).run(catch=False)
Traceback (most recent call last):
...
In conjunction with `%pdb` on, this allows for the debugger to jump directly to the first failure location.

**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
sage: tester = instance_tester(ZZ)

sage: tester.assertTrue(1 == 1)

sage: tester.assertTrue(1 == 0)
Traceback (most recent call last):
...  
AssertionError: False is not true

sage: tester.assertTrue(1 == 0, "this is expected to fail")
Traceback (most recent call last):
...  
AssertionError: this is expected to fail

sage: tester.assertEqual(1, 1)

sage: tester.assertEqual(1, 0)
Traceback (most recent call last):
...  
AssertionError: 1 != 0
```

The available assertion testing facilities are the same as in `unittest.TestCase [UNITTEST]`, which see (actually, by a slight abuse, tester is currently an instance of this class).

### 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 (in Python 2):

```python
    for _ in xrange(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!
```

```python
sage: test_add_commutes(10)
a == -1/2, b == -1/95 ...
Passes!
```

```python
sage: test_add_commutes(1000) # long time
```

7.1. Testing
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:

```python
sage: from sage.misc.random_testing import test_add_is_mul
```

We start by testing that we get reproducible results when setting `seed` to 0.

```python
sage: test_add_is_mul(2, verbose=True, seed=0)
a == -4, b == 0 ...
Random testing has revealed a problem in test_add_is_mul
Please report this bug! You may be the first person in the world to have seen this problem.
Please include this random seed in your bug report:
Random seed: 0
AssertionError()
```

Normally in a `@random_testing` doctest, we would leave off the `verbose=True` and the `# random`. We put it in here so that we can verify that we are seeing the exact same error when we reproduce the error below.

```python
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: 21639041059600942878250607128692114173
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”.

```python
sage: test_add_is_mul(10, verbose=True, seed=21639041059600942878250607128692114173)
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: 21639041059600942878250607128692114173
AssertionError()
```
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:

```
sage: from sage.misc.nested_class_test import TestParent1, TestParent2, TestParent3, TestParent4
sage: P = TestParent1()
sage: TestSuite(P).run()
Failure ...
The following tests failed: _test_elements, _test_pickling
```

They actually need to be in the NestedClassMetaclass. However, due to a technical detail, this is currently not directly supported:

```
sage: P = TestParent2()
Traceback (most recent call last):
  ...TypeError: metaclass conflict: the metaclass of a derived class must be a (non-strict)
  _subclass of the metaclasses of all its bases
sage: TestSuite(P).run()  # not tested
```

Instead, the easiest is to inherit from UniqueRepresentation, which is what you want to do anyway most of the time:

```
sage: P = TestParent3()
sage: TestSuite(P).run()
```

This is what all Sage’s parents using categories currently do. An alternative is to use ClasscallMetaclass as metaclass:

```
sage: P = TestParent4()
sage: TestSuite(P).run()
```

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)
```

```
sage.misc.benchmark.bench1()
Run a benchmark.

BENCHMARK:
```

7.2. Benchmarking and Profiling 397
sage: from sage.misc.benchmark import *
sage: print(bench1()[0])
Find the Mordell-Weil group of the elliptic curve 5077A using mwrank

sage.misc.benchmark.bench2()
Run a benchmark.

BENCHMARK:

sage: from sage.misc.benchmark import *
sage: print(bench2()[0])
Some basic arithmetic with very large Integer numbers: '3^1000001 * 19^100001

sage.misc.benchmark.bench3()
Run a benchmark.

BENCHMARK:

sage: from sage.misc.benchmark import *
sage: print(bench3()[0])
Some basic arithmetic with very large Rational numbers: '(2/3)^100001 * (17/19)^100001

sage.misc.benchmark.bench4()
Run a benchmark.

BENCHMARK:

sage: from sage.misc.benchmark import *
sage: print(bench4()[0])
Rational polynomial arithmetic using Sage. Compute (x^29+17*x-5)^200.

sage.misc.benchmark.bench5()
Run a benchmark.

BENCHMARK:

sage: from sage.misc.benchmark import *
sage: print(bench5()[0])
Rational polynomial arithmetic using Sage. Compute (x^19 - 18*x + 1)^50 one hundred times.

sage.misc.benchmark.bench6()
Run a benchmark.

BENCHMARK:

sage: from sage.misc.benchmark import *
sage: print(bench6()[0])
Compute the p-division polynomials of y^2 = x^3 + 37*x - 997 for primes p < 40.

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
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: \((x^97+19x+1)(x^{103}-19x^{97}+14)(x^{100}-1)\)
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^{1000001} * 19^{100001}
Time: ... seconds
Running benchmark 3
Some basic arithmetic with very large Rational numbers: '(2/3)^{100001} * (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
```

### 7.2. Benchmarking and Profiling
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
```

**AUTHOR:**

- William Stein, based on code by Fernando Perez included in IPython

**class** `sage.misc.sage_timeit.SageTimeitResult(stats, series=None)`

Bases: `object`

Represent the statistics of a timeit() command.

Prints as a string so that it can be easily returned to a user.

**INPUT:**

- `stats` – tuple of length 5 containing the following information:
  - integer, number of loops
  - integer, repeat number
  - Python integer, number of digits to print
  - number, best timing result
  - str, time unit

**EXAMPLES:**

```python
sage: from sage.misc.sage_timeit import SageTimeitResult
sage: SageTimeitResult( (3, 5, int(8), pi, 'ms') )
3 loops, best of 5: 3.1415927 ms per loop
```

```python
sage: units = [u"s", u"ms", u"μs", u"ns"]
sage: scaling = [1, 1e3, 1e6, 1e9]
sage: number = 7
sage: repeat = 13
sage: precision = int(5)
sage: best = pi / 10 ^ 9
sage: order = 3
sage: stats = (number, repeat, precision, best * scaling[order], units[order])
sage: SageTimeitResult(stats)
7 loops, best of 13: 3.1416 ns per loop
```

If the third argument is not a Python integer, a `TypeError` is raised:

```python
sage: SageTimeitResult( (1, 2, 3, 4, 's') )
<repr(<sage.misc.sage_timeit.SageTimeitResult at 0x...>) failed: TypeError: * wants
˓→int>
```

**sage.misc.sage_timeit.sage_timeit**

Accurately measure the wall time required to execute `stmt`.
INPUT:

- `stmt` – a text string.
- `globals_dict` – a dictionary or `None` (default). Evaluate `stmt` in the context of the globals dictionary. If not set, the current `globals()` dictionary is used.
- `preparse` – (default: use globals preparser default) if `True` preparse `stmt` using the Sage preparser.
- `number` – integer, (optional, default: 0), number of loops.
- `repeat` – integer, (optional, default: 3), number of repetition.
- `seconds` – boolean (default: `False`). Whether to just return time in seconds.

OUTPUT:

An instance of `SageTimeitResult` unless the optional parameter `seconds=True` is passed. In that case, the elapsed time in seconds is returned as a floating-point number.

EXAMPLES:

```sage
def junk():
    pass

sage: from sage.misc.sage_timeit import sage_timeit
sage: sage_timeit('3^100000', globals(), preparse=True, number=50)
  # random
˓→ output
50 loops, best of 3: 1.97 ms per loop
sage: sage_timeit('3^100000', globals(), preparse=False, number=50)
  # random
˓→ output
50 loops, best of 3: 67.1 ns per loop
sage: a = 10
sage: sage_timeit('a^2', globals(), number=50)
  # random
˓→ output
50 loops, best of 3: 4.26 us per loop
```

If you only want to see the timing and not have access to additional information, just use the `timeit` object:

```sage
sage: timeit('10^2', number=50)
50 loops, best of 3: ... per loop
```

Using `sage_timeit` gives you more information though:

```sage
def junk():
    pass

sage: from sage.misc.sage_timeit import sage_timeit
sage: from os import linesep as CR
sage: s = sage_timeit('a^2', globals(), repeat=1000)
sage: len(s.series)
1000
sage: mean(s.series)
# random output
3.1298141479492283e-07
sage: min(s.series)
# random output
2.9258728027343752e-07
sage: t = stats.TimeSeries(s.series)
sage: t.scale(10^6).plot_histogram(bins=20, figsize=[12,6], ymax=2)
```

The input expression can contain newlines (but doctests cannot, so we use `os.linesep` here):

```sage
sage: from sage.misc.sage_timeit import sage_timeit
sage: from os import import linesep as CR
sage: # sage_timeit('r^a = 2\nb=131\nfactor(a^b-1)')
```
sage: sage_timeit('a = 2 + CR + \''b=131'' + CR + \'factor(a^b-1)\'',
.....:    globals(), number=10)
10 loops, best of 3: ... per loop

Test to make sure that timeit behaves well with output:

sage: timeit("print('Hi')", number=50)
50 loops, best of 3: ... per loop

If you want a machine-readable output, use the seconds=True option:

sage: timeit("print('Hi')", seconds=True)  # random output
1.42555236816e-06
sage: t = timeit("print('Hi')", seconds=True)
sage: t  #r random output
3.6010742187499999e-07

### 7.2.3 The timeit command

This uses the function `sage_timeit()`.

```python
class sage.misc.sage_timeit_class.SageTimeit
    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 timeit module, which avoids a number of common traps for measuring execution times. It is also based on IPython’s %timeit command.

    TYPICAL INPUT FORMAT:

    timeit(statement, preparse=None, number=0, repeat=3, precision=3)

    EXAMPLES:

    sage: timeit('2^10000')
    625 loops, best of 3: ... per loop

    We illustrate some options:

    sage: timeit('2+2', precision=2, number=20, repeat=5)
    20 loops, best of 5: ... per loop

    The preparser is on by default (if it is on), but the preparse option allows us to override it:

    sage: timeit('2^10000', preparse=False, number=50)
    50 loops, best of 3: ... per loop

    The input can contain newlines:

    sage: timeit("a = 2\nb=131\nfactor(a^b-1)\", number=25)
    25 loops, best of 3: ... per loop
```
See also:

\texttt{runsnake()}

\texttt{eval}(\texttt{code, globs=None, locals=None, **kwds})

This eval function is called when doing \%timeit in the notebook.

INPUT:

\begin{itemize}
  \item code – string of code to evaluate; may contain newlines.
  \item globs – global variables; if not given, uses module scope globals.
  \item locals – ignored completely.
  \item kwds – passed onto sage\_timeit. Common options are \texttt{preparse}, \texttt{number}, \texttt{repeat}, \texttt{precision}. See \texttt{sage\_timeit()} for details.
\end{itemize}

OUTPUT: string – timing information as a string

EXAMPLES:

\begin{verbatim}
sage: timeit.eval("2+2")                      # random output
 '625 loops, best of 3: 1.47 us per loop'
\end{verbatim}

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.

\begin{verbatim}
sage: timeit('sleep(float(0.5))', number=3)  # long time (5s on sage.math, 2012)
3 loops, best of 3: ... ms per loop
\end{verbatim}

7.2.4 Simple profiling tool

AUTHORS:

\begin{itemize}
  \item David Harvey (August 2006)
  \item Martin Albrecht
\end{itemize}

\textbf{class} \texttt{sage.misc.profiler.Profiler}(\texttt{systems=[], verbose=False})

Bases: \texttt{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:

\begin{verbatim}
sage: def f():                                # not tested
...:   p = Profiler()                        # not tested
\end{verbatim}

Calling \texttt{p(message)} creates a checkpoint:

\begin{verbatim}
sage: p("try factoring 15")                  # not tested
\end{verbatim}

Do something time-consuming:

\begin{verbatim}
sage: x = factor(15)                         # not tested
\end{verbatim}

You can create a checkpoints without a string; \texttt{Profiler} will use the source code instead:

```
sage: p()  # not tested
sage: y = factor(25)  # not tested
sage: p("last step")  # not tested
sage: z = factor(35)  # not tested
sage: p()  # not tested
```

This will give a nice list of timings between checkpoints:

```
sage: print(p)  # not tested
```

Let’s try it out:

```
sage: f()  # not tested
   3.020s -- try factoring 15
   15.240s -- line 17: y = factor(25)
   5000.190s -- last step
```

See also:

`runsnake()`

Todo:

• Add Pyrex source code inspection (I assume it doesn’t currently do this)
• Add ability to sort output by time
• Add option to constructor to print timing immediately when checkpoint is reached
• Migrate to Pyrex?
• Add ability to return timings in a more machine-friendly format

AUTHOR:

• David Harvey (August 2006)

```
clear()
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:

```
sage: from sage.misc.gperftools import Profiler, run_100ms
sage: prof = Profiler()
sage: prof.start()  # optional - gperftools
sage: run_100ms()
sage: prof.stop()  # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
```
REFERENCE:

Uses the Google performance analysis tools. Note that they are not included in Sage, you have to install them yourself on your system.

AUTHORS:

• Volker Braun (2014-03-31): initial version

class sage.misc.gperftools.Profiler(filename=None)

Bases: sage.structure.sage_object.SageObject

Interface to the gperftools profiler

INPUT:

• filename – string or None (default). The file name to log to. By default, a new temporary file is created.

EXAMPLES:

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage: from sage.misc.gperftools import Profiler</td>
<td>Profiler</td>
</tr>
<tr>
<td>sage: Profiler()</td>
<td>Profiler logging to ...</td>
</tr>
</tbody>
</table>

filename()

Return the file name

OUTPUT:

String.

EXAMPLES:

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage: from sage.misc.gperftools import Profiler</td>
<td>Profiler</td>
</tr>
<tr>
<td>sage: prof = Profiler()</td>
<td></td>
</tr>
<tr>
<td>sage: prof.filename()</td>
<td>'./tmp_.../perf'</td>
</tr>
</tbody>
</table>

save(filename, cumulative=True, verbose=True)

Save report to disk.

INPUT:

• filename – string. The filename to save at. Must end with one of .dot, .ps, .pdf, .svg, .gif, or .txt to specify the output file format.

• cumulative – boolean (optional, default: True). Whether to return cumulative timings.

• verbose – boolean (optional, default: True). Whether to print informational messages.

EXAMPLES:

<table>
<thead>
<tr>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage: from sage.misc.gperftools import Profiler, run_100ms</td>
<td>Profiler, run_100ms</td>
</tr>
<tr>
<td>sage: prof = Profiler()</td>
<td></td>
</tr>
<tr>
<td>sage: prof.start()</td>
<td># optional - gperftools</td>
</tr>
<tr>
<td>sage: run_100ms()</td>
<td># optional - gperftools</td>
</tr>
<tr>
<td>sage: prof.stop()</td>
<td># optional - gperftools</td>
</tr>
<tr>
<td>PROFILES: interrupts/evictions/bytes = ...</td>
<td></td>
</tr>
<tr>
<td>sage: f = tmp_filename(ext='.txt')</td>
<td># optional - gperftools</td>
</tr>
<tr>
<td>sage: prof.save(f, verbose=False)</td>
<td># optional - gperftools</td>
</tr>
</tbody>
</table>
**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
sage: from sage.misc.gperftools import Profiler, run_100ms
sage: prof = Profiler()
sage: prof.start()  # optional - gperftools
sage: run_100ms()
sage: prof.stop()   # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
```

**top(cumulative=True)**

Print text report

OUTPUT:

Nothing. A textual report is printed to stdout.

EXAMPLES:

```python
sage: from sage.misc.gperftools import Profiler
sage: prof = Profiler()
sage: prof.start()  # optional - gperftools
sage: # do something
sage: prof.stop()   # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
sage: prof.top()   # optional - gperftools
Using local file ...
Using local file ...
```

**sage.misc.gperftools.crun(s, evaluator)**

Profile single statement.

- **s** – string. Sage code to profile.
- **evaluator** – callable to evaluate.

EXAMPLES:

```python
sage: import sage.misc.gperftools as gperf
sage: ev = lambda ex:eval(ex, globals(), locals())
sage: gperf.crun('gperf.run_100ms()', evaluator=ev)  # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
Using local file ...
Using local file ...
```
sage.misc.gperftools.run_100ms()

A function that performs some computation for more than (but not that much more than) 100ms.

EXAMPLES:

```
sage: from sage.misc.gperftools import run_100ms
sage: run_100ms()
```

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:

```
sage: from sage.docs.instancedoc import instancedoc
sage: @instancedoc
....: class X(object):
....:     "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:

```
sage: cython(''
....: from sage.docs.instancedoc import instancedoc
....: cdef class Y:
....:     "Class docstring"
....:     def _instancedoc_(self):
....:         return "Instance docstring"
....:     instancedoc(Y)
....:     '''
sage: Y.__doc__
'File:...
Class docstring'
sage: Y().__doc__
'Instance docstring'
```
One can still add a custom __doc__ attribute on a particular instance:

```python
sage: obj = X()
sage: obj.__doc__ = "Very special doc"
sage: print(obj.__doc__)
Very special doc
```

This normally does not work on extension types:

```python
sage: Y().__doc__ = "Very special doc"
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:

```python
sage: @instancedoc
class Meta(type):
    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
```

```python
class sage.docs.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.docs.instancedoc import InstanceDocDescriptor
    sage: def instancedoc(self):
    ....:     return "Instance doc"
    sage: docattr = InstanceDocDescriptor("Class doc", instancedoc)
    sage: class Z(object):
    ....:     __doc__ = InstanceDocDescriptor("Class doc", instancedoc)
    sage: Z.__doc__
    'Class doc'
    sage: Z().__doc__
    'Instance doc'
```

We can still override the __doc__ attribute of the instance:
sage: obj = Z()
sage: obj.__doc__ = "Custom doc"
sage: obj.__doc__
'Custom doc'
sage: del obj.__doc__
sage: obj.__doc__
'Instance doc'

sage.docs.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.

AUTHORS:
  • originally taken from Fernando Perez’s IPython
  • William Stein (extensive modifications)
  • Nick Alexander (extensions)
  • Nick Alexander (testing)
  • Simon King (some extension for Cython, generalisation of SageArgSpecVisitor)

EXAMPLES:

```sage
def from sage.misc.sageinspect import *
```

Test introspection of modules defined in Python and Cython files:

Cython modules:

```sage
def sage_getfile(sage.rings.rational)
'.../rational.pyx'
sage
def sage_getdoc(sage.rings.rational).lstrip()
'Rational Numbers...'
sage
def sage_getsource(sage.rings.rational)
'# distutils: ...Rational Numbers...'
```

Python modules:
Test introspection of classes defined in Python and Cython files:

Cython classes:

```
sage: sage_getfile(sage.rings.rational.Rational)  
'/.../rational.pyx'
sage: sage_getdoc(sage.rings.rational.Rational).lstrip()  
'A rational number...'
sage: sage_getsource(sage.rings.rational.Rational)  
'cdef class Rational...'
```

Python classes:

```
sage: sage_getfile(BlockFinder)  
'/.../sage/misc/sageinspect.py'
sage: sage_getdoc(BlockFinder).lstrip()  
'Provide a tokeneater() method to detect the...'
sage: sage_getsource(BlockFinder)  
'class BlockFinder:...'
```

Test introspection of functions defined in Python and Cython files:

Cython functions:

```
sage: sage_getdef(sage.rings.rational.make_rational, obj_name='mr')  
'mr(s)'
sage: sage_getfile(sage.rings.rational.make_rational)  
'/.../rational.pyx'
sage: sage_getdoc(sage.rings.rational.make_rational).lstrip()  
'Make a rational number ...'
sage: sage_getsource(sage.rings.rational.make_rational)[4:]  
'make_rational(s):...'
```

Python functions:

```
sage: sage_getdef(sage.misc.sageinspect.sage_getfile, obj_name='sage_getfile')  
'sage_getfile(obj)'
```
sage: sage_getfile(sage.misc.sageinspect.sage_getfile)
'.../sageinspect.py'
sage: sage_getdoc(sage.misc.sageinspect.sage_getfile).lstrip()
'Get the full file name associated to "obj" as a string...'
sage: sage_getsource(sage.misc.sageinspect.sage_getfile)[4:]
'sage_getfile(obj):...'

Unfortunately, no argspec is extractable from builtins. Hence, we use a generic argspec:
sage: sage_getdef(''.find, 'find')
'find(*args, **kwd')
sage: sage_getdef(str.find, 'find')
'find(*args, **kwd')

By trac ticket #9976 and trac ticket #14017, introspection also works for interactively defined Cython code, and with rather tricky argument lines:
sage: cython('def foo(unsigned int x=1, a=\'\'', b={not (2+1==3):\'bar\'}, *args, **kwds): return')
sage: print(sage_getsource(foo))
def foo(unsigned int x=1, a=''''), b={not (2+1==3):'bar'}, *args, **kwds): return
sage: sage_getargspec(foo)
ArgSpec(args=['x', 'a', 'b'], varargs='args', keywords='kwds', defaults=(1, '``'), **{False: 'bar'})

class sage.misc.sageinspect.BlockFinder
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(type, token, srow_scol, erow_ecol, line)

class sage.misc.sageinspect.SageArgSpecVisitor
Bases: ast.NodeVisitor

A simple visitor class that walks an abstract-syntax tree (AST) for a Python function’s argspec. It returns the contents of nodes representing the basic Python types: None, booleans, numbers, strings, lists, tuples, and dictionaries. We use this class in _sage_getargspec_from_ast() to extract an argspec from a function’s or method’s source code.

EXAMPLES:
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: visitor.visit(ast.parse('[1,2,3]').body[0].value)
[1, 2, 3]
sage: v = visitor.visit(ast.parse("\"a":\'e',2,[None,({False:True},'pi')]\"), 37.0: 
\"temp\"\").body[0].value
sage: sorted(v.items(), key=lambda x: str(x[0]))
[(37.0, 'temp'), ('a', ('e', 2, [None, ((False:True), 'pi')])))

(continues on next page)
sage: v = ast.parse("jc = ['veni', 'vidi', 'vici']").body[0]; v
<...ast.Assign object at ...

sage: attrs = [x for x in dir(v) if not x.startswith('_')]

sage: v._attributes in attrs and v._fields in attrs and v.col_offset in attrs
True

sage: visitor.visit(v.targets[0])
'jc'

sage: visitor.visit(v.value)
['veni', 'vidi', 'vici']

\textbf{visit\_BinOp}(node)

Visit a Python AST \texttt{ast.BinOp} node.

INPUT:

\begin{itemize}
\item node - the node instance to visit
\end{itemize}

OUTPUT:

\begin{itemize}
\item The result that node represents
\end{itemize}

AUTHOR:

\begin{itemize}
\item Simon King
\end{itemize}

EXAMPLES:

\begin{verbatim}
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']]
\end{verbatim}

\textsc{\#indirect doctest } # py2
[11, 15, 6, 2, 2, 48]

\begin{verbatim}
sage: [vis(d) for d in ['(3+(2*4))', '7|8', '5^3', '7/3', '7//3', '3<<4']]
\end{verbatim}

\textsc{\#indirect doctest } # py3
[11, 15, 6, 2.3333333333333335, 2, 48]

\textbf{visit\_BoolOp}(node)

Visit a Python AST \texttt{ast.BoolOp} node.

INPUT:

\begin{itemize}
\item node - the node instance to visit
\end{itemize}

OUTPUT:

\begin{itemize}
\item The result that node represents
\end{itemize}

AUTHOR:

\begin{itemize}
\item Simon King
\end{itemize}

EXAMPLES:

\begin{verbatim}
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit(ast.parse(x).body[0].value)
sage: [vis(d) for d in ['True and 1', 'False or 3 or None', '3 and 4']]
\end{verbatim}

\textsc{\#indirect doctest }
[1, 3, 4]
visit_Compare(node)
Visit a Python AST `ast.Compare` node.

INPUT:
- `node` - the node instance to visit

OUTPUT:
- The result that `node` represents

AUTHOR:
- Simon King

EXAMPLES:
```python
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_Compare(ast.parse(x).body[0].value)
sage: [vis(d) for d in ['1<2==2!=3', '1==1>2', '1<2>1', '1<3<2<4']]
[True, False, True, False]
```

visit_Dict(node)
Visit a Python AST `ast.Dict` node.

INPUT:
- `node` - the node instance to visit

OUTPUT:
- the dictionary the `node` represents

EXAMPLES:
```python
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}']
sage: [[sorted(d.items(), key=lambda x: str(x[0])) for d in v]
[[[], [(1, 'one'), ('other', 'bother'), ('two', 2)]]
```

visit_List(node)
Visit a Python AST `ast.List` node.

INPUT:
- `node` - the node instance to visit

OUTPUT:
- the list the `node` represents

EXAMPLES:
```python
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']]]
```
visit_Name(node)
Visit a Python AST ast.Name node.

INPUT:
• node - the node instance to visit

OUTPUT:
• None, True, False, or the node's name as a string.

EXAMPLES:
```
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_Name(ast.parse(x).body[0].value)
[sage: [vis(n) for n in ['True', 'False', 'None', 'foo', 'bar']]  # py2
[True, False, None, 'foo', 'bar']
sage: [type(vis(n)) for n in ['True', 'False', 'None', 'foo', 'bar']]  # py2
[type 'bool', type 'bool', type 'NoneType', type 'str', type 'str']
sage: [vis(n) for n in ['foo', 'bar']]  # py3
['foo', 'bar']
sage: [type(vis(n)) for n in ['foo', 'bar']]  # py3
[type 'str', type 'str']
```

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  # py3
sage: visitor = sms.SageArgSpecVisitor()  # py3
sage: vis = lambda x: visitor.visit_NameConstant(ast.parse(x).body[0].value)  # py3
sage: [vis(n) for n in ['True', 'False', 'None']]  # py3
[True, False, None]
sage: [type(vis(n)) for n in ['True', 'False', 'None']]  # py3
[type 'bool', type 'bool', type 'NoneType']
```

visit_Num(node)
Visit a Python AST ast.Num node.

INPUT:
• node - the node instance to visit

OUTPUT:
• the number the node represents

EXAMPLES:
```
```
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_Num(ast.parse(x).body[0].value)
sage: [vis(n) for n in ['123', '0.0', str(-pi.n())]]  # py2
[123, 0.0, -3.14159265358979]
sage: [vis(n) for n in ['123', '0.0']]  # py3
[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
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_Str(ast.parse(x).body[0].value)
sage: [vis(s) for s in ['abstract', 'u\'syntax', 'r''r\'tr\ee''']] ['abstract', u'syntax', 'tr\ee']
```

**visit_Tuple(node)**
Visit a Python AST `ast.Tuple` node.

**INPUT:**
- node - the node instance to visit

**OUTPUT:**
- the tuple the node represents

**EXAMPLES:**

```python
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_Tuple(ast.parse(x).body[0].value)
sage: [vis(t) for t in [(), (x,y), ('Au', 'Al', 'Cu')]] [(x', 'y'), ('Au', 'Al', 'Cu')]
```

**visit_UnaryOp(node)**
Visit a Python AST `ast.BinOp` node.

**INPUT:**
- node - the node instance to visit

**OUTPUT:**
- The result that node represents
• Simon King

EXAMPLES:

```
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit_UnaryOp(ast.parse(x).body[0].value)
sage: [vis(d) for d in ['+(3*2)', '-(3*2)']]
[6, -6]
```

`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:**

```
sage: import ast, sage.misc.sageinspect as sms  # py3
sage: s = "def f(a, b=2, c={"a": [4, 5.5, False]}, d=(None, True)): \n    return \n...
  "  # py3
sage: visitor = sms.SageArgSpecVisitor()  # py3
sage: args = ast.parse(s).body[0].args.args  # py3
sage: [visitor.visit_arg(n) for n in args]  # py3
["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` (optional, default None)

This is only relevant with Python 3, so the doctests are marked accordingly.

**EXAMPLES:**

```
sage: from sage.misc.sageinspect import formatannotation
sage: import inspect
sage: def foo(a, *, b:int, **kwargs):  # py3
    pass
....
  s = inspect.signature(foo)  # py3
sage: a = s.parameters['a'].annotation  # py3
sage: a  # py3
<class 'inspect._empty'>
sage: formatannotation(a)  # py3
'inspect._empty'
```

(continues on next page)
```python
sage: b = s.parameters['b'].annotation # py3
sage: b # py3
<class 'int'>
sage: formatannotation(b) # py3
'int'
```

```python
sage.misc.sageinspect.isclassinstance(obj)
Checks 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
sage: isclassinstance(myclass2)
False
```

```python
sage.misc.sageinspect.loadable_module_extension()
Return the filename extension of loadable modules, including the dot.

It is ‘.dll’ on cygwin, ‘.so’ otherwise.

EXAMPLES:
```python
sage: from sage.misc.sageinspect import loadable_module_extension
sage: sage.structure.sage_object.__file__.endswith(loadable_module_extension())
True
```

```python
sage.misc.sageinspect.sage_formatargspec(args, varargs=None, varkw=None, defaults=None, kwonlyargs=(), kwonlydefaults={}, annotations=(), formatarg=<class 'str'>, formatvarargs=<function <lambda> at 0x7fe48d509550>, formatvarkw=<function <lambda> at 0x7fe48d5095e0>, formatvalue=<function <lambda> at 0x7fe48d509670>, formatreturns=<function <lambda> at 0x7fe48d509700>, formatannotation=<function formatannotation at 0x7fe48d5094c0>)

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.ArgSpec`.

**EXAMPLES:**
```
sage: from sage.misc.sageinspect import sage_formatargspec
dsage: from inspect import formatargspec  # deprecated in Python 3
sage: args = ['a', 'b', 'c']
sage: defaults = [3]
sage: sage_formatargspec(args, defaults=defaults)
'(a, b, c=3)'
sage: import warnings; warnings.simplefilter('ignore')  # py3: ignore
˓→DeprecationWarning
sage: formatargspec(args, defaults=defaults) == sage_formatargspec(args, ...
˓→defaults=defaults)
True
```

`sage.misc.sageinspect.sage_getargspec(obj)`

Return the names and default values of a function’s arguments.

**INPUT:**

obj, any callable object

**OUTPUT:**

An `ArgSpec` is returned. This is a named tuple `(args, varargs, keywords, defaults)`.

- `args` is a list of the argument names (it may contain nested lists).
- `varargs` and `keywords` are the names of the `*` and `**` arguments or `None`.
- `defaults` is an `n`-tuple of the default values of the last `n` arguments.

**NOTE:**

If the object has a method `__sage_argspec__` then the output of that method is transformed into a named tuple and then returned.

If a class instance has a method `__sage_src__` then its output is studied to determine the argspec. This is because currently the `CachedMethod` decorator has no `__sage_argspec__` method.

**EXAMPLES:**
```
sage: from sage.misc.sageinspect import sage_getargspec
sage: def f(x, y, z=1, t=2, *args, **keywords):
    ....:     pass
sage: sage_getargspec(f)
ArgSpec(args=['x', 'y', 'z', 't'], varargs='args', keywords='keywords', defaults=(1, ˓→2))
```

We now run `sage_getargspec` on some functions from the Sage library:
```
sage: sage_getargspec(identity_matrix)
ArgSpec(args=['ring', 'n', 'sparse'], varargs=None, keywords=None, defaults=(0, ˓→False))
sage: sage_getargspec(factor)
ArgSpec(args=['n', 'proof', 'int_', 'algorithm', 'verbose'], varargs=None, keywords=˓→'kwds', defaults=(None, False, 'pari', 0))
```
In the case of a class or a class instance, the `ArgSpec` of the `__new__`, `__init__` or `__call__` method is returned:

```
sage: P.<x,y> = QQ[]
sage: sage_getargspec(P)
ArgSpec(args=['base_ring', 'n', 'names', 'order'], varargs=None, keywords=None, defaults=('degrevlex',))
sage: sage_getargspec(P.__class__)
ArgSpec(args=['self', 'x'], varargs='args', keywords='kwds', defaults=(0,))
```

The following tests against various bugs that were fixed in trac ticket #9976:

```
sage: from sage.rings.polynomial.real_roots import bernstein_polynomial_factory_ratlist
sage: sage_getargspec(bernstein_polynomial_factory_ratlist.coeffs_bitsize)
ArgSpec(args=['self'], varargs=None, keywords=None, defaults=None)
sage: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
sage: sage_getargspec(BooleanMonomialMonoid.gen)
ArgSpec(args=['self', 'i'], varargs=None, keywords=None, defaults=(0,))
sage: I = P*[x,y]
sage: sage_getargspec(I.groebner_basis)
ArgSpec(args=['self', 'algorithm', 'deg_bound', 'mult_bound', 'prot'], varargs='args', keywords='kwds', defaults=('degrevlex', ()), keywords_only=False)
sage: cython("cpdef int foo(x,y) except -1: return 1")
sage: sage_getargspec(foo)
ArgSpec(args=['x', 'y'], varargs=None, keywords=None, defaults=None)
```

If a `functools.partial` instance is involved, we see no other meaningful solution than to return the argspec of the underlying function:

```
sage: def f(a,b,c,d=1):
    ....:     return a+b+c+d
sage: import functools
sage: f1 = functools.partial(f, 1,c=2)
sage: sage_getargspec(f1)
ArgSpec(args=['a', 'b', 'c', 'd'], varargs=None, keywords=None, defaults=(1,))
```

AUTHORS:

- William Stein: a modified version of inspect.getargspec from the Python Standard Library, which was taken from IPython for use in Sage.
- Extensions by Nick Alexander
- Simon King: Return an `ArgSpec`, fix some bugs.

```
sage.misc.sageinspect.sage_getdef(obj, obj_name='')
```

Return the definition header for any callable object.

INPUT:

- `obj` - function
- `obj_name` - string (optional, default '')

`obj_name` is prepended to the output.

EXAMPLES:
Check that trac ticket #6848 has been fixed:

```sage
sage: sage_getdef(RDF.random_element)
'(min=-1, max=1)'
```

If an exception is generated, None is returned instead and the exception is suppressed.

AUTHORS:
- William Stein
- extensions by Nick Alexander

`sage.misc.sageinspect.sage_getdoc(obj, obj_name='', embedded=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:

```sage
sage: from sage.misc.sageinspect import sage_getdoc
sage: sage_getdoc(identity_matrix)[87:124]
'Return the n x n identity matrix over'
sage: def f(a,b,c,d=1): return a+b+c+d
...  
sage: import functools
sage: f1 = functools.partial(f, 1,c=2)
sage: f.__doc__ = "original documentation"
sage: f1.__doc__ = "specialised documentation"
sage: sage_getdoc(f)
'original documentation\n'
sage: sage_getdoc(f1)
'specialised documentation\n'
```

AUTHORS:
- William Stein
- extensions by Nick Alexander

`sage.misc.sageinspect.sage_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:

```python
from sage.misc.sageinspect import sage_getdoc_original
```

Here is a class that has its own docstring:

```python
print(sage_getdoc_original(sage.rings.integer.Integer))
```

The `Integer` class represents arbitrary precision integers. It derives from the `Element` class, so integers can be used as ring elements anywhere in Sage.

If the class does not have a docstring, the docstring of the `__init__` method is used, but not the `__init__` method of the base class (this was fixed in trac ticket #24936):

```python
from sage.categories.category import Category
class A(Category):
    def __init__(self):
        '''The __init__ docstring'''
class B(Category):
    pass
print(sage_getdoc_original(A))
The __init__ docstring'
sage_getdoc_original(B) ''
```

Old-style classes are supported:

```python
class OldStyleClass:
    def __init__(self):
        '''The __init__ docstring'''
    pass
print(sage_getdoc_original(OldStyleClass))
The __init__ docstring
```

When there is no `__init__` method, we just get an empty string:

```python
class OldStyleClass:
    pass
print(sage_getdoc_original(OldStyleClass)) ''
```

If an instance of a class does not have its own docstring, the docstring of its class results:

```python
sage_getdoc_original(sage.plot.colors.aliceblue) == sage_getdoc_original(sage.plot.colors.Color)
True
```

`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:
The following tests against some bugs fixed in trac ticket #9976:

```python
sage: obj = sage.combinat.partition_algebra.SetPartitionsAk
sage: obj = sage.combinat.partition_algebra.SetPartitionsAk
sage: sage_getfile(obj)
'...sage/combinat/partition_algebra.py'
```

And here is another bug, fixed in trac ticket #11298:

```python
sage: P.<x,y> = QQ[
```

A problem fixed in trac ticket #16309:

```python
sage: cython('''
.....: class Bar: pass
.....: cdef class Foo: pass
.....: ''')
```

By trac ticket #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)
''
```

AUTHORS:
- Nick Alexander
- Simon King

`sage.misc.sageinspect.sage_getsource(obj)`
Return the source code associated to obj as a string, or None.

INPUT:
- obj – function, etc.

EXAMPLES:
AUTHORS:

• William Stein

• extensions by Nick Alexander

**sage.misc.sageinspect.sage_getsourcelines**(obj)

Return a pair ([source_lines], starting line number) of the source code associated to obj, or None.

**INPUT:**

• obj – function, etc.

**OUTPUT:**

(source_lines, lineno) or None: source_lines is a list of strings, and lineno is an integer.

**EXAMPLES:**

```python
sage: from sage.misc.sageinspect import sage_getsourcelines
sage: sage_getsourcelines(matrix)[1]
21
sage: sage_getsourcelines(matrix)[0][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`:

```python
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(object):
'
```

AUTHORS:

• William Stein

• Extensions by Nick Alexander

• Extension to interactive Cython code by Simon King

• Simon King: If a class has no docstring then let the class definition be found starting from the __init__ method.

• Simon King: Get source lines for dynamic classes.

**sage.misc.sageinspect.sage_getvariablename**(self, omit_underscore_names=True)

Attempt to get the name of a Sage object.

**INPUT:**

• self – any object.

• omit_underscore_names – boolean, default True.

**OUTPUT:**

If the user has assigned an object obj to a variable name, then return that variable name. If several variables point to obj, return a sorted list of those names. If omit_underscore_names is True (the default) then omit names starting with an underscore “_”.

This is a modified version of code taken from http://pythonic.pocoo.org/2009/5/30/finding-objects-names, written by Georg Brandl.
EXAMPLES:

```
sage: from sage.misc.sageinspect import sage_getvariablename
sage: A = random_matrix(ZZ, 100)
sage: sage_getvariablename(A)
'A'
sage: B = A
sage: sage_getvariablename(A)
['A', 'B']
```

If an object is not assigned to a variable, an empty list is returned:

```
sage: sage_getvariablename(random_matrix(ZZ, 60))
[]
```

### 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 `EDITOR` is set to a known editor, then the system will use that if no template has been set explicitly.

```
sage.misc.edit_module.edit(obj, editor=None, bg=None)
```

Open source code of `obj` in editor of your choice.

**INPUT:**

- `editor` – str (default: None); If given, use specified editor. Choice is stored for next time.

**AUTHOR:**

Nils Bruin (2007-10-03)

**EXAMPLES:**

This is a typical example of how to use this routine:

```
# make some object obj
sage: edit(obj)  # not tested
```

Now for more details and customization:

```
sage: import sage.misc.edit_module as m
sage: m.set_edit_template("vi -c ${line} ${file}")
```
In fact, since \texttt{vi} is a well-known editor, you could also just use:

\begin{verbatim}
sage: m.set_editor("vi")
\end{verbatim}

To illustrate:

\begin{verbatim}
sage: m.edit_template.template
'vi -c \${line} \${file}'
\end{verbatim}

And if your environment variable \texttt{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:

\begin{verbatim}
sage: edit(edit)
# not tested
\end{verbatim}

\begin{verbatim}
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:
\end{verbatim}

\begin{verbatim}
sage: %edit gcd # indirect doctest, not tested
sage: %ed gcd # indirect doctest, not tested
\end{verbatim}

The above should open your favorite editor (as stored in the environment variable \texttt{EDITOR}) with the file in which
gcd is defined, and when your editor supports it, also at the line in which gcd is defined.

\begin{verbatim}
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 -b’). 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 \texttt{sageinspect} to get the file and the line.
EXAMPLES:
\end{verbatim}

\begin{verbatim}
sage: import sage.misc.edit_module as edit_module
sage: edit_module.file_and_line(sage)
('...sage/__init__.py', 0)
\end{verbatim}

The following tests against a bug that was fixed in trac ticket \#11298:

\begin{verbatim}
sage: edit_module.file_and_line(x)
('...sage/symbolic/expression...pyx', ...)
\end{verbatim}

\begin{verbatim}
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’. How-
ever, if the editor set in the shell variable \texttt{EDITOR} is known, then the system will substitute an appropriate template
for you. See edit_module.template_defaults for the recognised templates.

7.3. Miscellaneous Inspection and Development Tools
AUTHOR:
Nils Bruin (2007-10-03)
EXAMPLES:

```python
sage: from sage.misc.edit_module import set_edit_template
sage: set_edit_template("echo EDIT ${file}:${line}"
)  # not tested
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:

```python
sage: from sage.misc.edit_module import set_editor
sage: set_editor('vi')
sage: sage.misc.edit_module.edit_template.template
'vi -c ${line} ${file}'
```

`sage.misc.edit_module.template_fields(template)`

Given a String.Template object, returns the fields.

AUTHOR:
Nils Bruin (2007-10-22)
EXAMPLES:

```python
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']
```

### 7.3.4 Get resource usage of process

AUTHORS:
- William Stein (2006-03-04): initial version
- Jeroen Demeyer (2016-11-14): implement as thin wrapper over psutil package

`sage.misc.getusage.get_memory_usage(t=None)``

Return the memory usage of the current process in megabytes.

INPUT:
- `t` – a float (default: None); output of an earlier call. If this is given, return the current usage minus `t`.

OUTPUT: a float representing the number of megabytes used.

EXAMPLES:
sage.misc.getusage.virtual_memory_limit()
Return the upper limit for virtual memory usage.
This is the value set by ulimit -v at the command line or a practical limit if no limit is set. In any case, the value is bounded by sys.maxsize.

OUTPUT:
Integer. The virtual memory limit in bytes.

EXAMPLES:

sage: from sage.misc.getusage import virtual_memory_limit
sage: virtual_memory_limit() > 0
True
sage: virtual_memory_limit() <= sys.maxsize
True

7.3.5 Class inheritance graphs

sage.misc.classgraph.class_graph(top, depth=5, name_filter=None, classes=None, as_graph=True)
Return the class inheritance graph of a module, class, or object

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

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

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

sage: from sage.rings.polynomial.padics import polynomial_padic_capped_relative_dense, polynomial_padic_capped_relative_flatt
sage: G = class_graph(sage.rings.polynomial.padics); G
Digraph on 6 vertices
sage: G.vertices()
['Polynomial',
 'Polynomial_generic_cdv',
 'Polynomial_generic_dense',
 'Polynomial_padic',
 'Polynomial_padic_capped_relative_dense',
 (continues on next page)
We construct the inheritance graph of a given class:

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

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

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

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

```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']}
```

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

**Note:** `class_graph` does not yet handle nested classes

### 7.3.6 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 obj is an instance of a class.

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

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

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

If `module_name` is `None`, the function runs through all loaded modules and returns the list of objects whose name matches `name`.

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

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

EXAMPLES:

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

```python
sage: import sympy
```

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

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

```python
sage: dt.find_objects_from_name('RR', '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
```

```python
sage: objs = dt.find_objects_from_name('find_objects_from_name')
sage: len(objs)
1
```

```python
sage: dt.find_objects_from_name is dt.find_objects_from_name
True
```

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)`

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:

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

`sage.misc.dev_tools.import_statements(*objects, **kwds)`
Print import statements for the given objects.

**INPUT:**

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

**EXAMPLES:**

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

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

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

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

```python
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 ZZ
sage: import_statements(euler_phi)
from sage.arith.misc import euler_phi
```

(continues on next page)
If you don't like the warning you can disable them with the option `verbose`:

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

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

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

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

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

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

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

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:

```python
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)
load sage.monoids.automatic_semigroup... succeeded
load sage.monoids.hecke_monoid... succeeded
load sage.monoids.indexed_free_monoid... succeeded
sage: import_statements('HeckeMonoid')
from sage.monoids.hecke_monoid import HeckeMonoid

We test different objects which have no appropriate answer:

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

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

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

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

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

The following were fixed with trac ticket #15351:

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

(continues on next page)
Check that a name from the global namespace is properly found (see trac ticket #23779):

```python
sage: import_statements('log')
from sage.functions.log import log
```

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

---

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

Load all submodules of a given 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:

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

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

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

The second argument allows to exclude a pattern:

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

```python
sage: sage.misc.dev_tools.runsnake(command)
```

Graphical profiling with runsnake

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

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

command is first preparsed (see preparses):

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

1
4
9

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

```bash
> 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

### 7.3.7 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,

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

but from the perspective of a wrapper, they are different:
```python
sage: def wrap(g):
....    def _g(*args,**kwargs):
....        print("{} {}".format(args, kwargs))
....        return g(*args, **kwargs)
....    return _g
sage: h = wrap(f)

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)

f
```

```text
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
```
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

\[
\text{fix_to_pos}(\ldots) \quad \text{self.f(*A,**dict(K))}
\]

and

\[
\text{self.f}(\ldots)
\]

in all cases.

**EXAMPLES:**

```python
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))))
```

```python
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

\[
A,K = self.fix_to_pos(\ldots)
\text{self.f(*A,**dict(K))}
\]

are equivalent to

\[
\text{self.f}(\ldots)
\]

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:**
7.3.8 ReST index of functions

This module contains a function that generates a ReST index table of functions for use in doc-strings.

**gen_rest_table_index()**  
Return a ReST table describing a list of functions.

```
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"
```

```
sage.misc.rest_index_of_methods.gen_rest_table_index(obj, names=None, sort=True, only_local_functions=True)
Return a ReST table describing a list of functions.
The list of functions can either be given explicitly, or implicitly as the functions/methods of a module or class.
In the case of a class, only non-inherited methods are listed.

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; True) – whether to sort the list of methods lexicographically.
```
• only_local_functions (boolean; 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.

**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:**

```python
sage: from sage.misc.rest_index_of_methods import gen_rest_table_index
sage: print(gen_rest_table_index([graphs.PetersenGraph]))
```

![](image1.png)

The table of a module:

```python
sage: print(gen_rest_table_index(sage.misc.rest_index_of_methods))
```

![](image2.png)

The table of a class:

```python
sage: print(gen_rest_table_index(Graph))
```

![](image3.png)

```python
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 function (or methods) of a module (or class).

**INPUT:**
• root – the module, or class, whose elements are to be listed.

• additional_categories – 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; 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]
sage: Graph.bipartite_color in l
True
```

sage.misc.rest_index_of_methods.list_of_subfunctions(root, only_local_functions=True)
Returns 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; 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]
sage: Graph.bipartite_color in l
True
```
8.1 Low-level memory allocation functions

AUTHORS:

• Jeroen Demeyer (2011-01-13): initial version (trac ticket #10258)
• Jeroen Demeyer (2014-12-14): add more functions (trac ticket #10257)
• Jeroen Demeyer (2015-03-02): move from c_lib to Cython (trac ticket #17881)

```
sage.ext.memory.init_memory_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()
```

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.

```
sage.misc.c3.C3_algorithm(start, bases, attribute, proper)
```
An implementation of the C3 algorithm.

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

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

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

INPUT:

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

**ASSUMPTIONS:**

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

**OUTPUT:**

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

**EXAMPLES:**

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

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

sage: from sage.misc.c3 import C3_algorithmsage: class HierarchyElement(UniqueRepresentation):
  ....: @lazy_attribute
  ....: def all_bases(self):
  ....:     return C3_algorithm(self, '_bases', '_all_bases', False)
  ....: def __repr__(self):
  ....:     return self._name
  ....: def __init__(self, name, bases):
  ....:     self._name = name
  ....:     self._bases = list(bases)

We construct a little hierarchy:

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

And inspect the linear extensions associated to each element:

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

So far so good. However:

```python
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 \texttt{Foo}, since \texttt{A} and \texttt{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(object): pass
sage: class A2(object):
    ....:    def foo(self): return 2
sage: class A3(object): pass
sage: class A4(object):
    ....:    def foo(self): return 4
sage: class A5(A2, A1):
    ....:    def foo(self): return 5
sage: class A6(A4, A3): pass
sage: class A7(A6, A5): pass
```

If \texttt{a} is an instance of \texttt{A7}, then Python needs to choose which implementation to use upon calling \texttt{a.foo()}: that of \texttt{A4} or \texttt{A5}, but obviously not that of \texttt{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 \texttt{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):

```python
sage: class B6(A1,A2): pass
sage: class B7(B6,A5): pass # py2
Traceback (most recent call last):
 ...
TypeError: Error when calling the metaclass bases
  Cannot create a consistent method resolution order (MRO) for bases ...
```

```python
sage: class B7(B6,A5): pass # py3
Traceback (most recent call last):
 ...
TypeError: Cannot create a consistent method resolution order (MRO) for bases A1, A2
```

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']
```

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']
```

### 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 super classes of a class as its bases. However, this would have several drawbacks:

- It is not so elegant, in particular because it duplicates information: we already know through A5 that A7 is a subclass of A1. This duplication could be acceptable in our context because the hierarchy of classes is generated automatically from a conceptual hierarchy (the categories) which serves as single point of truth for calculating the bases of each class.

- It increases the complexity of the calculation of the MRO with C3. For example, for a linear 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 \(\text{dir}\) 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 \(\text{dir}\) for a typical class like \(\text{GradedHopfAlgebrasWithBasis(QQ).parent_class}\) with 37 super classes took 18 seconds with this approach.

Granted: this mostly affects the \(\text{dir}\) 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 \(\text{instrumented C3}\)) 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 \(\text{controlled C3}\)) yields the desired MRO.

**EXAMPLES:**

As an experimentation and testing tool, we use a class \(\text{HierarchyElement}\) 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], 6: [3, 2], 5: [3, 1], 4: [2, 1]}, linear_extension=True, facade=True)
```

And build a \(\text{HierarchyElement}\) from it:

```python
```

8.3. The C3 algorithm, under control of a total order
```python
def from sage.misc.c3_controlled import HierarchyElement
def x = HierarchyElement(10, P)
```

Here are its bases:

```python
def HierarchyElement(10, P).bases
[9, 8, 7]
```

Using the standard C3 algorithm fails:

```python
def x.mro_standard
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 $P$:

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

Now we play with a specific linear extension of $P$:

```python
def Q = P.linear_extension([10, 9, 8, 7, 6, 5, 4, 1, 2, 3]).to_poset()
def 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]]
def x = HierarchyElement(10, Q)
def 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
def x.mro
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
def x.mro_controlled
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

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

```python
def x._test_mro()
```

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

```python
def x._bases
[9, 8, 7]
def x._bases_controlled
[9, 8, 7, 6, 5]
```

Altogether, four bases were added for control:
This information can also be recovered with:

\[
\text{sage: } \text{x.all_bases_len()}
\]
15
\[
\text{sage: } \text{x.all_bases_controlled_len()}
\]
19

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

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

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

\[
\text{sage: } \text{sorted(Word(stats).evaluation_sparse())}
\]

\[[(1, 36), (2, 108), (3, 180), (4, 216), (5, 180)]\]

We now consider a hierarchy of categories:

\[
\text{sage: from operator import attrgetter}
\]
\[
\text{sage: x = HierarchyElement(Groups(), attrcall("super_categories"), attrgetter("_cmp_key __"))}
\]

\[
\text{sage: x.mro}
\]

[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]

\[
\text{sage: x.mro_standard}
\]

[Category of groups, Category of monoids, Category of semigroups, Category of inverse unital magmas, Category of unital magmas, Category of magmas, Category of sets, Category of sets with partial maps, Category of objects]

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

\[
\text{sage: } C = \text{FiniteFields()}
\]
\[
\text{sage: x = HierarchyElement(C, attrcall("super_categories"), attrgetter("_cmp_key"))}
\]
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:

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

AUTHOR:

sage.misc.c3_controlled.C3_merge(lists)
Return the input lists merged using the C3 algorithm.

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

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

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

sage.misc.c3_controlled.C3_sorted_merge(lists, key=’identity’)
Return the sorted input lists merged using the C3 algorithm, with a twist.
INPUT:

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

OUTPUT: a pair (result, suggestion)

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

EXAMPLES:

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

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

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

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

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

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

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

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

And indeed C3_merge() now returns the desired result:

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

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

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

Base cases:

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

8.3. The C3 algorithm, under control of a total order 449
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]])
([4,3,2,1], [4,3,2,1])
sage: C3_sorted_merge_check([[1],[3,2],[[]]])
([3,2,1], [2,1])
sage: C3_sorted_merge_check([[1],[4,3,2],[2]])
([4,3,2,1], [2,1])
sage: C3_sorted_merge_check([[2],[1],[4],[3]])
([4,3,2,1], [4,2,1])
sage: C3_sorted_merge_check([[2],[1],[4],[[]]])
([4,3,2,1], [4,2,1])
sage: C3_sorted_merge_check([[2],[1],[3],[4]])
([4,3,2,1], [4,3,2,1])
sage: C3_sorted_merge_check([[2],[1],[3,2,1],[3]])
([3,2,1], [3])
sage: C3_sorted_merge_check([[2],[1],[2,1],[3]])
([3,2,1], [3,2])
```

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

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

```python
class sage.misc.c3_controlled.CmpKey
Bases: object

This class implements the lazy attribute Category._cmp_key.
```
The comparison key \texttt{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 \(A._\text{cmp}\_\text{key} > B._\text{cmp}\_\text{key}\). 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 \(A._\text{cmp}\_\text{key} > B._\text{cmp}\_\text{key}\)). In particular, \texttt{Objects()} is the largest category.

- If \(A! = B\) and taking the join of \(A\) and \(B\) makes sense (e.g. taking the join of \texttt{Algebras(GF(5))} and \texttt{Algebras(QQ)} does not make sense), then \(A < B\) or \(B < A\).

The rationale for the inversion above between \(A < B\) and \(A._\text{cmp}\_\text{key} > B._\text{cmp}\_\text{key}\) 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 ticket \#13589 and would be reasonably session independent. The number \(i\) is there to resolve ambiguities; it is session dependent, and is assigned increasingly when new categories are created.

\textbf{Note:} This is currently not implemented using a \texttt{lazy\_attribute} for speed reasons only (the code is in Cython and takes advantage of the fact that Category objects always have a \texttt{\_dict\_} dictionary)

\textbf{Todo:}

- Handle nicely (covariant) functorial constructions and axioms

\textbf{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
```

(continues on next page)
(96, ...)
sage: Crystals()._cmp_key
(136, ...)
sage: AdditiveMagmas()._cmp_key
(256, ...)
sage: Magmas()._cmp_key
(4096, ...)
sage: CommutativeAdditiveSemigroups()._cmp_key
(256, ...)
sage: Rings()._cmp_key
(225536, ...)
sage: Algebras(QQ)._cmp_key
(225536, ...)
sage: AlgebrasWithBasis(QQ)._cmp_key
(227584, ...)
sage: GradedAlgebras(QQ)._cmp_key
(226560, ...)
sage: GradedAlgebrasWithBasis(QQ)._cmp_key
(228608, ...)

For backward compatibility we currently want the following comparisons:

```
sage: EnumeratedSets()._cmp_key > Sets().Facade()._cmp_key
True
sage: AdditiveMagmas()._cmp_key > EnumeratedSets()._cmp_key
True
sage: Category.join([EnumeratedSets(), Sets().Facade()]).parent_class._an_element_._
˓→module__
'sage.categories.enumerated_sets'
sage: GradedAlgebras(QQ)._cmp_key < Magmas()._cmp_key
True
sage: VectorSpaces(QQ)._cmp_key < Rings()._cmp_key
True
sage: VectorSpaces(QQ)._cmp_key < Magmas()._cmp_key
True
```

class sage.misc.c3_controlled.CmpKeyNamed
    Bases: object

    This class implements the lazy attribute CategoryWithParameters._cmp_key.

    See also:

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

    Note:

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

EXAMPLES:

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

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

Bases: object

A class for elements in a hierarchy.

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

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

INPUT:

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

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

EXAMPLES:

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

```python
sage: from sage.misc.c3_controlled import HierarchyElement
sage: G = DiGraph({
    ....: 44 : [43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27,
    ....: 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6,
    ....: 5, 4, 3, 2, 1, 0],
    ....: 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],
    ....: 42 : [36, 35, 37, 30, 29, 28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
    ....: 41 : [40, 36, 35, 33, 32, 31, 30, 29, 28, 27, 26, 23, 22, 21, 20, 19, 18,
    ....: 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
    ....: 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],
    ....: 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],
    ....: 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],
})
```
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

Chapter 8. Low-Level Utilities
all_bases()
Return the set of all instances of HierarchyElement above self, self included.

EXAMPLES:

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

all_bases_controlled_len()
Return the cumulated size of the controlled bases of the elements above self in the hierarchy.

EXAMPLES:

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

all_bases_len()
Return the cumulated size of the bases of the elements above self in the hierarchy.

EXAMPLES:

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

bases()
The bases of self.

The bases are given as a list of instances of HierarchyElement, sorted decreasingly according to the key function.

EXAMPLES:
from sage.misc.c3_controlled import HierarchyElement
P = Poset((divisors(30), lambda x,y: y.divides(x)), facade=True)
x = HierarchyElement(10, P)
x.bases
[5, 2]
type(x.bases[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:
from sage.misc.c3_controlled import HierarchyElement
P = Poset((divisors(30), lambda x,y: y.divides(x)), facade=True)
x = HierarchyElement(1, P)
x.cls
<class '1.cls'>
x.cls.mro()
[<class '1.cls'>, <... 'object'>]
x = HierarchyElement(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 $C_3$-sorted-merge().

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

mro_controlled()
The MRO for this object, calculated with $C_3$-merge(), under control of $C_3$-sorted-merge()
EXAMPLES:

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

```python
sage.misc.c3_controlled.identity(x)
```

EXAMPLES:

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

INDICES AND TABLES

- Index
- Module Index
- Search Page
a
sage.arith.srange, 80

d
sage.databases.sql_db, 92
sage.docs.instancedoc, 407

e
sage.ext.fast_callable, 202
sage.ext.fast_eval, 219
sage.ext.memory, 441

f
sage.features, 374
sage.features.bliss, 380
sage.features.csdp, 381
sage.features.databases, 381
sage.features.fes, 382
sage.features.gap, 382
sage.features.graph_generators, 383
sage.features.linalg, 384

m
sage.media.wav, 115
sage.misc.abstract_method, 49
sage.misc.banner, 349
sage.misc.benchmark, 397
sage.misc.bindable_class, 52
sage.misc.c3, 441
sage.misc.c3_controlled, 443
sage.misc.cachefunc, 166
sage.misc.call, 68
sage.misc.callable_dict, 68
sage.misc.citation, 384
sage.misc.citation, 384
sage.misc.classcall_metaclass, 61
sage.misc.classgraph, 427
sage.misc.constant_function, 60
sage.misc.converting_dict, 69
sage.misc.copying, 385
sage.misc.cython, 229
sage.misc.decorators, 54
sage.misc.defaults, 1

sage.misc.dev_tools, 428
sage.misc.dist, 373
sage.misc.edit_module, 424
sage.misc.element_with_label, 83
sage.misc.explain_pickle, 327
sage.misc.fast_methods, 65
sage.misc.flatten, 72
sage.misc.fpickle, 326
sage.misc.func_persist, 315
sage.misc.function_mangling, 434
sage.misc.functional, 2
sage.misc.getusage, 426
sage.misc.gperftools, 404
sage.misc.html, 288
sage.misc.inherit_comparison, 63
sage.misc.inline_fortran, 232
sage.misc.latex, 293
sage.misc.latex_macros, 312
sage.misc.lazy_attribute, 144
sage.misc.lazy_format, 154
sage.misc.lazy_import, 155
sage.misc.lazy_import_cache, 159
sage.misc.lazy_list, 159
sage.misc.lazy_string, 165
sage.misc.map_threaded, 79
sage.misc.mathml, 292
sage.misc.messaging, 89
sage.misc.method_decorator, 65
sage.misc.misc, 122
sage.misc.misc_c, 135
sage.misc.mrange, 73
sage.misc.multireplace, 79
sage.misc.nested_class, 339
sage.misc.nested_class_test, 397
sage.misc.object_multiplexer, 65
sage.misc.package, 366
sage.misc.pager, 356
sage.misc.parser, 233
sage.misc.persist, 315
sage.misc.prange, 41
sage.misc.profiler, 403
sage.misc.python, 242

461
sage.misc.random_testing, 394
sage.misc.randstate, 30
sage.misc.remote_file, 88
sage.misc.repr, 261
sage.misc.reset, 351
sage.misc.rest_index_of_methods, 437
sage.misc.sage_eval, 243
sage.misc.sage_input, 263
sage.misc.sage_ostools, 90
sage.misc.sage_timeit, 400
sage.misc.sage_timeit_class, 402
sage.misc.sage_unittest, 387
sage.misc.sagedoc, 356
sage.misc.sageinspect, 409
sage.misc.search, 73
sage.misc.session, 344
sage.misc.sh, 246
sage.misc.sphinxify, 366
sage.misc.stopgap, 118
sage.misc.superseded, 119
sage.misc.table, 283
sage.misc.temporary_file, 84
sage.misc.unknown, 47
sage.misc.verbose, 142
sage.misc.viewer, 353
sage.misc.weak_dict, 193

S
sage.structure.graphics_file, 113

T
sage.typeset.ascii_art, 255
sage.typeset.character_art, 250
sage.typeset.character_art_factory, 252
sage.typeset.symbols, 247
sage.typeset.unicode_art, 259
sage.typeset.unicode_characters, 260
INDEX

Symbols

_rich_repr_ (sage.misc.html.HtmlFragment method), 290
_rich_repr_ (sage.misc.table.table method), 287

A

abs() (sage.ext.fast_callable.Expression method), 206
abs() (sage.ext.fast_eval.FastDoubleFunc method), 221
absolute_path() (sage.features.StaticFile method), 379
abstract_method (in module sage.misc.abstract_method), 49
abstract_methods_of_class (in module sage.misc.abstract_method), 52
AbstractMethod (class in sage.misc.abstract_method), 49
add_column() (sage.databases.sql_db.SQLDatabase method), 96
add_data() (sage.databases.sql_db.SQLDatabase method), 97
add_library() (sage.misc.inline_fortran.InlineFortran method), 232
add_library_path() (sage.misc.inline_fortran.InlineFortran method), 232
add_macro() (sage.misc.latex.Latex method), 294
add_package_to_preamble_if_available() (sage.misc.latex.Latex method), 294
add_row() (sage.databases.sql_db.SQLDatabase method), 97
add_rows() (sage.databases.sql_db.SQLDatabase method), 97
add_to_preamble() (sage.misc.latex.Latex method), 294
additive_order() (in module sage.misc.functional), 5
all_bases() (sage.misc.c3_controlled.HierarchyElement method), 455
all_bases_controlled_len() (sage.misc.c3_controlled.HierarchyElement method), 455
all_bases_len() (sage.misc.c3_controlled.HierarchyElement method), 455
APPEND() (sage.misc.explain_pickle.PickleExplainer method), 329
append() (sage.misc.explain_pickle.TestAppendList method), 334
APPENDS() (sage.misc.explain_pickle.PickleExplainer method), 330
arccos() (sage.ext.fast_eval.FastDoubleFunc method), 221
arccosh() (sage.ext.fast_eval.FastDoubleFunc method), 221
arcsin() (sage.ext.fast_eval.FastDoubleFunc method), 221
arcsinh() (sage.ext.fast_eval.FastDoubleFunc method), 222
arctan() (sage.ext.fast_eval.FastDoubleFunc method), 222
arctanh() (sage.ext.fast_eval.FastDoubleFunc method), 222
ArgumentFixer (class in sage.misc.function_mangling), 434
arguments() (sage.ext.fast_callable.ExpressionCall method), 207
ascii_art() (in module sage.typeset.ascii_art), 257
AsciiArt (class in sage.typeset.ascii_art), 257
assign() (sage.misc.sage_input.SageInputBuilder method), 270
atomic_write (class in sage.misc.temporary_file), 84
attrcall() (in module sage.misc.call), 68
AttrCallObject (class in sage.misc.call), 68

B

BackslashOperator (class in sage.misc.misc), 122
backtrack() (sage.misc.parser.Tokenizer method), 239
balanced_sum() (in module sage.misc.functional), 136
banner() (in module sage.misc.banner), 349
banner_text() (in module sage.misc.banner), 349
base() (sage.ext.fast_callable.ExpressionIPow method), 209
base_field() (in module sage.misc.functional), 5
base_ring() (in module sage.misc.functional), 5
bases() (sage.misc.c3_controlled.HierarchyElement method), 455
Index

464

basis() (in module sage.misc.functional), 5
bench0() (in module sage.misc.benchmark), 397
bench1() (in module sage.misc.benchmark), 397
bench2() (in module sage.misc.benchmark), 398
bench3() (in module sage.misc.benchmark), 398
bench4() (in module sage.misc.benchmark), 398
bench5() (in module sage.misc.benchmark), 398
bench6() (in module sage.misc.benchmark), 398
bench7() (in module sage.misc.benchmark), 398
benchmark() (in module sage.misc.benchmark), 399
benchmark_libc() (in module sage.misc.randstate), 37
benchmark_mt() (in module sage.misc.randstate), 37
Benzene (class in sage.features.graph_generators), 383
betavariate() (in module sage.misc.prandom), 42
BindableClass (class in sage.misc.bindable_class), 52
BINFLOAT() (sage.misc.explain_pickle.PickleExplainer method), 330
BINGET() (sage.misc.explain_pickle.PickleExplainer method), 330
BININT() (sage.misc.explain_pickle.PickleExplainer method), 330
BININT1() (sage.misc.explain_pickle.PickleExplainer method), 330
BININT2() (sage.misc.explain_pickle.PickleExplainer method), 330
BINPERSID() (sage.misc.explain_pickle.PickleExplainer method), 330
BININPUT() (sage.misc.explain_pickle.PickleExplainer method), 330
BINSTRING() (sage.misc.explain_pickle.PickleExplainer method), 330
BINUNICODE() (sage.misc.explain_pickle.PickleExplainer method), 330
blackboard_bold() (sage.misc.latex.Latex method), 295
Bliss (class in sage.features.bliss), 380
BlissLibrary (class in sage.features.bliss), 380
BlockFinder (class in sage.misc.sageinspect), 411
bool_function() (in module sage.misc.latex), 302
bool_function() (in module sage.misc.mathml), 292
BoundClass (class in sage.misc.bindable_class), 54
browser() (in module sage.misc.viewer), 354
browser() (sage.misc.viewer.Viewer method), 353
Buckygen (class in sage.features.graph_generators), 383
BUILD() (sage.misc.explain_pickle.PickleExplainer method), 330
build() (sage.typeset.character_art_factory.CharacterArtFactory method), 252
build_from_magic_method() (sage.typeset.character_art_factory.CharacterArtFactory method), 253
build_from_string() (sage.typeset.character_art_factory.CharacterArtFactory method), 253
build_list() (sage.typeset.character_art_factory.CharacterArtFactory method), 254
build_set() (sage.typeset.character_art_factory.CharacterArtFactory method), 254
build_tuple() (sage.typeset.character_art_factory.CharacterArtFactory method), 254
builtin_constant_function() (in module sage.misc.latex), 303
by_opcode (sage.ext.fast_callable.InterpreterMetadata attribute), 214
by_opname (sage.ext.fast_callable.InterpreterMetadata attribute), 215

C
C3_algorithm() (in module sage.misc.c3), 441
C3_merge() (in module sage.misc.c3_controlled), 448
C3_sorted_merge() (in module sage.misc.c3_controlled), 448
c_rand_double() (sage.misc.randstate.randstate method), 38
c_random() (sage.misc.randstate.randstate method), 38
cache (sage.misc.cachefunc.CachedFunction attribute), 174
cache() (sage.misc.sage_input.SageInputBuilder method), 270
cache_key() (in module sage.misc.cachefunc), 187
cached() (sage.misc.cachefunc.CachedFunction attribute), 174
cached() (sage.misc.cachefunc.CachedMethodCaller method), 179
cached_function() (in module sage.misc.cachefunc), 188
cached_in_parent_method() (in module sage.misc.cachefunc), 189
cached_method() (in module sage.misc.cachefunc), 189
CachedFunction (class in sage.misc.cachefunc), 173
CacheDict (class in sage.misc.cachefunc), 173
CachedInParentMethod (class in sage.misc.cachefunc), 177
CachedMethod (class in sage.misc.cachefunc), 177
CachedMethodCaller (class in sage.misc.cachefunc), 179
CachedMethodCallerNoArgs (class in sage.misc.cachefunc), 180
CachedMethodPickle (class in sage.misc.cachefunc), 182
CachedSpecialMethod (class in sage.misc.cachefunc), 183
DatabaseCremona (class in sage.features.databases), 381
DatabaseJones (class in sage.features.databases), 381
DatabaseKnotInfo (class in sage.features.databases), 382
db() (in module sage.misc.persist), 318
db_save() (in module sage.misc.persist), 318
decomposition() (in module sage.misc.functional), 7
decorator_defaults() (in module sage.misc.decorators), 55
decorator_keywords() (in module sage.misc.decorators), 55
default_rows() (sage.databases.sql_db.SQLDatabase method), 99
delete_tmpfiles() (in module sage.misc.temporary_file), 87
denominator() (in module sage.misc.functional), 8
deprecated_function_alias() (in module sage.misc.superseded), 119
DeprecationWarning (class in sage.misc.superseded), 119
deprecation() (in module sage.misc.superseded), 120
dest_fd (sage.misc sage_ostools.redirected attribute), 91
dest_file (sage.misc sage_ostools.redirected attribute), 91
det() (in module sage.misc.functional), 8
detex() (in module sage.misc.sagedoc), 356
DICT() (sage.misc.sage_input.SageInputBuilder method), 330
dict() (sage.misc.sage_input.SageInputBuilder method), 271
dict_function() (in module sage.misc.latex), 303
dict_key() (in module sage.misc.cachefunc), 191
dim() (in module sage.misc.functional), 8
dimension() (in module sage.misc.functional), 8
disc() (in module sage.misc.functional), 9
discriminant() (in module sage.misc.functional), 9
disk_cached_function (class in sage.misc.cachefunc), 191
DiskCachedFunction (class in sage.misc.cachefunc), 183
doc_index() (in module sage.misc.rest_index_of_methods), 437
domain() (sage.media.wav.Wave method), 116
drop_column() (sage.databases.sql_db.SQLDatabase method), 99
drop_data_from_table() (sage.databases.sql_db.SQLDatabase method), 100
drop_index() (sage.databases.sql_db.SQLDatabase method), 100
drop_primary_key() (sage.databases.sql_db.SQLDatabase method), 101
drop_table() (sage.databases.sql_db.SQLDatabase method), 101
drop_unique() (sage.databases.sql_db.SQLDatabase method), 101
dummy() (sage.misc.nested_class.MainClass.NestedClass.NestedSubClass method), 343
dumps() (in module sage.misc.persist), 318
dumps() (sage.misc.persist.SagePickler class method), 316
DUP() (sage.misc.explain_pickle.PickleExplainer method), 330
dup_source_fd (sage.misc.sage_ostools.redirected attribute), 91
dvi_viewer() (in module sage.misc.viewer), 355
dvi_viewer() (sage.misc.viewer.Viewer method), 353

e
edit() (in module sage.misc.edit_module), 424
effect() (in module sage.misc.edit_module), 425
EDITOR, 424, 425
ElementWithLabel (class in sage.misc.element_with_label), 83
ellipsis_ITER() (in module sage.arith.srange), 80
ellipsis_RANGE() (in module sage.arith.srange), 80
empty() (sage.typeset.character_art.CharacterArt class method), 250
EMPTY_DICT() (sage.misc.explain_pickle.PickleExplainer method), 330
EMPTY_LIST() (sage.misc.explain_pickle.PickleExplainer method), 330
empty_subscript() (sage.misc.sage_input.SageInputBuilder method), 272
EMPTY_TUPLE() (sage.misc.explain_pickle.PickleExplainer method), 330
EmptyNewstyleClass (class in sage.misc.explain_pickle), 329
EmptyOldstyleClass (class in sage.misc.explain_pickle), 329
evaluate() (sage.misc.latex.Latex method), 295
evaluate() (sage.misc.latex.Latex method), 295
evaluate() (sage.misc.latex.Latex method), 295
evaluate() (sage.misc.latex.Latex method), 295
environment variable
EDITOR, 424, 425
PATH, 90
eta() (in module sage.misc.functional), 9
eval() (sage.misc.html.HTMLFragmentFactory method), 288
eval() (sage.misc.html.MathJax method), 290
eval() (sage.misc.inline_fortran.InlineFortran method), 232
eval() (sage.misc.latex.Latex method), 296
eval() (sage.misc.python.Python method), 242
eval() (sage.misc.sage_timeit_class.SageTimeit method), 403
eval() (sage.misc.sh.Sh method), 246
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exactly_one_is_true()</code></td>
<td><code>sage.misc.misc</code></td>
<td>124</td>
</tr>
<tr>
<td><code>Executable()</code></td>
<td><code>sage.features</code></td>
<td>375</td>
</tr>
<tr>
<td><code>exists()</code></td>
<td><code>sage.misc.misc</code></td>
<td>125</td>
</tr>
<tr>
<td><code>exp()</code></td>
<td><code>sage.ext.fast_eval.FastDoubleFunc</code></td>
<td>223</td>
</tr>
<tr>
<td><code>experimental()</code></td>
<td><code>sage.misc.superseded</code></td>
<td>120</td>
</tr>
<tr>
<td><code>experimental_packages()</code></td>
<td><code>sage.misc.package</code></td>
<td>367</td>
</tr>
<tr>
<td><code>experimentalWarning()</code></td>
<td><code>sage.misc.superseded</code></td>
<td>121</td>
</tr>
<tr>
<td><code>explain_pickle()</code></td>
<td><code>sage.misc.explain_pickle.PickleExplainer</code></td>
<td>336</td>
</tr>
<tr>
<td><code>explain_pickle_string()</code></td>
<td><code>sage.misc.explain_pickle.PickleExplainer</code></td>
<td>337</td>
</tr>
<tr>
<td><code>exponent()</code></td>
<td><code>sage.ext.fast_callable.ExpressionIPow</code></td>
<td>209</td>
</tr>
<tr>
<td><code>exppovariate()</code></td>
<td><code>sage.misc.prandom</code></td>
<td>42</td>
</tr>
<tr>
<td><code>Expression()</code></td>
<td><code>sage.ext.fast_callable</code></td>
<td>206</td>
</tr>
<tr>
<td><code>ExpressionCall()</code></td>
<td><code>sage.ext.fast_callable</code></td>
<td>207</td>
</tr>
<tr>
<td><code>ExpressionChoice()</code></td>
<td><code>sage.ext.fast_callable</code></td>
<td>207</td>
</tr>
<tr>
<td><code>ExpressionConstant()</code></td>
<td><code>sage.ext.fast_callable</code></td>
<td>208</td>
</tr>
<tr>
<td><code>ExpressionIPow()</code></td>
<td><code>sage.ext.fast_callable</code></td>
<td>209</td>
</tr>
<tr>
<td><code>ExpressionTreeBuilder()</code></td>
<td><code>sage.ext.fast_callable</code></td>
<td>209</td>
</tr>
<tr>
<td><code>ExpressionVariable()</code></td>
<td><code>sage.ext.fast_callable</code></td>
<td>211</td>
</tr>
<tr>
<td><code>EXT1()</code></td>
<td><code>sage.misc.explain_pickle.PickleExplainer</code></td>
<td>330</td>
</tr>
<tr>
<td><code>EXT2()</code></td>
<td><code>sage.misc.explain_pickle.PickleExplainer</code></td>
<td>330</td>
</tr>
<tr>
<td><code>EXT4()</code></td>
<td><code>sage.misc.explain_pickle.PickleExplainer</code></td>
<td>330</td>
</tr>
<tr>
<td><code>extend()</code></td>
<td><code>sage.misc.explain_pickle.TestAppendList</code></td>
<td>334</td>
</tr>
<tr>
<td><code>extension()</code></td>
<td><code>sage.structure.graphics_file.Mime</code></td>
<td>114</td>
</tr>
<tr>
<td><code>extra_macros()</code></td>
<td><code>sage.misc.latex.Latex</code></td>
<td>297</td>
</tr>
<tr>
<td><code>extra_preamble()</code></td>
<td><code>sage.misc.latex.Latex</code></td>
<td>297</td>
</tr>
<tr>
<td><code>f()</code></td>
<td><code>sage.misc.cachefunc.CachedFunction</code></td>
<td>175</td>
</tr>
<tr>
<td><code>f() </code></td>
<td><code>sage.misc.function_mangling.ArgumentFixer</code></td>
<td>435</td>
</tr>
<tr>
<td><code>fast_callable()</code></td>
<td><code>sage.ext.fast_callable</code></td>
<td>215</td>
</tr>
<tr>
<td><code>fast_float()</code></td>
<td><code>sage.ext.fast_eval.FastDoubleFunc</code></td>
<td>225</td>
</tr>
<tr>
<td><code>fast_float_arg()</code></td>
<td><code>sage.ext.fast_eval.FastDoubleFunc</code></td>
<td>226</td>
</tr>
<tr>
<td><code>fast_float_constant()</code></td>
<td><code>sage.ext.fast_eval.FastDoubleFunc</code></td>
<td>226</td>
</tr>
<tr>
<td><code>fast_float_func()</code></td>
<td><code>sage.ext.fast_eval.FastDoubleFunc</code></td>
<td>226</td>
</tr>
<tr>
<td><code>FastDoubleFunc()</code></td>
<td><code>sage.ext.fast_eval.FastDoubleFunc</code></td>
<td>226</td>
</tr>
<tr>
<td><code>FastHashable_class()</code></td>
<td><code>sage.misc.fast_methods</code></td>
<td>65</td>
</tr>
<tr>
<td><code>fcp()</code></td>
<td><code>sage.misc.functional</code></td>
<td>9</td>
</tr>
<tr>
<td><code>Feature()</code></td>
<td><code>sage.features</code></td>
<td>376</td>
</tr>
<tr>
<td><code>FeatureNotPresentError</code></td>
<td></td>
<td>377</td>
</tr>
<tr>
<td><code>FeatureTestResult()</code></td>
<td><code>sage.features</code></td>
<td>377</td>
</tr>
<tr>
<td><code>file_and_line()</code></td>
<td><code>sage.misc.edit_module</code></td>
<td>425</td>
</tr>
<tr>
<td><code>file_list()</code></td>
<td><code>sage.misc.cachefunc.FileCache</code></td>
<td>184</td>
</tr>
<tr>
<td><code>FileCache()</code></td>
<td><code>sage.misc.cachefunc</code></td>
<td>184</td>
</tr>
<tr>
<td><code>filename()</code></td>
<td><code>sage.misc.gperftools.Profiler</code></td>
<td>405</td>
</tr>
<tr>
<td><code>filename()</code></td>
<td><code>sage.structure.graphics_file.GraphicsFile</code></td>
<td>113</td>
</tr>
<tr>
<td><code>find_object_modules()</code></td>
<td><code>sage.misc.dev_tools</code></td>
<td>428</td>
</tr>
<tr>
<td><code>find_objects_from_name()</code></td>
<td><code>sage.misc.dev_tools</code></td>
<td>429</td>
</tr>
<tr>
<td><code>finish_startup()</code></td>
<td><code>sage.misc.lazy_import</code></td>
<td>156</td>
</tr>
<tr>
<td><code>fix_to_named()</code></td>
<td><code>sage.misc.function_mangling.ArgumentFixer</code></td>
<td>435</td>
</tr>
<tr>
<td><code>fix_to_pos()</code></td>
<td><code>sage.misc.function_mangling.ArgumentFixer</code></td>
<td>436</td>
</tr>
<tr>
<td><code>flatten()</code></td>
<td><code>sage.misc.flatten</code></td>
<td>72</td>
</tr>
<tr>
<td><code>FLOAT()</code></td>
<td><code>sage.misc.explain_pickle.PickleExplainer</code></td>
<td>304</td>
</tr>
<tr>
<td><code>float_function()</code></td>
<td><code>sage.misc.latex</code></td>
<td>330</td>
</tr>
<tr>
<td><code>float_str()</code></td>
<td><code>sage.misc.sagedoc.SageInputBuilder</code></td>
<td>272</td>
</tr>
<tr>
<td><code>floor()</code></td>
<td><code>sage.ext.fast_eval.FastDoubleFunc</code></td>
<td>223</td>
</tr>
<tr>
<td><code>foo()</code></td>
<td><code>sage.misc.parser</code></td>
<td>242</td>
</tr>
<tr>
<td><code>forall()</code></td>
<td><code>sage.misc.misc</code></td>
<td>125</td>
</tr>
<tr>
<td><code>format()</code></td>
<td><code>sage.misc.sagedoc.SageInputFormatter</code></td>
<td>357</td>
</tr>
<tr>
<td><code>format()</code></td>
<td><code>sage.misc.sagedoc.SageInputFormatter</code></td>
<td>279</td>
</tr>
<tr>
<td><code>format_search_as_html()</code></td>
<td><code>sage.misc.sagedoc</code></td>
<td>357</td>
</tr>
<tr>
<td><code>format_src()</code></td>
<td><code>sage.misc.sagedoc</code></td>
<td>358</td>
</tr>
<tr>
<td><code>formatannotation()</code></td>
<td><code>sage.misc.sageinspect</code></td>
<td>416</td>
</tr>
<tr>
<td><code>func_persist()</code></td>
<td><code>sage.misc.functional</code></td>
<td>315</td>
</tr>
<tr>
<td><code>function()</code></td>
<td><code>sage.ext.fast_callable.ExpressionCall</code></td>
<td>207</td>
</tr>
<tr>
<td><code>function_name()</code></td>
<td><code>sage.ext.fast_callable</code></td>
<td>217</td>
</tr>
<tr>
<td><code>Gamma()</code></td>
<td></td>
<td>43</td>
</tr>
<tr>
<td><code>GapPackage()</code></td>
<td><code>sage.features.gap</code></td>
<td>382</td>
</tr>
<tr>
<td><code>gauss()</code></td>
<td><code>sage.misc.prandom</code></td>
<td>43</td>
</tr>
<tr>
<td><code>gen()</code></td>
<td><code>sage.misc.functional</code></td>
<td>9</td>
</tr>
<tr>
<td><code>gen()</code></td>
<td><code>sage.misc.sagedoc.SageInputBuilder</code></td>
<td>220</td>
</tr>
</tbody>
</table>
krull_dimension() (in module sage.misc.functional), 14
last() (sage.misc.parser.Tokenizer method), 240
last_token_string() (sage.misc.parser.Tokenizer method), 240
Latex (class in sage.misc.latex), 293
latex() (in module sage.misc.latex), 305
latex_extra_preamble() (in module sage.misc.latex), 306
latex_variable_name() (in module sage.misc.latex), 306
latex_variable_names() (in module sage.misc.defaults), 1
latex_varify() (in module sage.misc.latex), 307
LatexCall (class in sage.misc.latex), 300
LatexExamples (class in sage.misc.latex), 300
LatexExamples.diagram (in class sage.misc.latex), 301
LatexExamples.graph (class in sage.misc.latex), 301
LatexExamples.knot (class in sage.misc.latex), 301
LatexExamples.pstricks (class in sage.misc.latex), 301
LatexExpr (class in sage.misc.latex), 301
launch_viewer() (sage.structure.graphics_file.GraphicsFile method), 113
lazy_attribute (class in sage.misc.lazy_attribute), 144
lazy_class_attribute (class in sage.misc.lazy_attribute), 152
lazy_import() (in module sage.misc.lazy_import), 156
lazy_list() (in module sage.misc.lazy_list), 161
lazy_list_formatter() (in module sage.misc.lazy_list), 162
lazy_list_from_function (class in sage.misc.lazy_list), 163
lazy_list_from_iterator (class in sage.misc.lazy_list), 163
lazy_list_from_update_function (class in sage.misc.lazy_list), 163
lazy_list_generic (class in sage.misc.lazy_list), 164
lazy_string() (in module sage.misc.lazy_string), 165
LazyFormat (class in sage.misc.lazy_format), 154
LazyImport (class in sage.misc.lazy_import), 156
LibFES (class in sage.features.fes), 382
LibFESLibrary (class in sage.features.fes), 382
License (class in sage.misc.copying), 385
lift() (in module sage.misc.functional), 14
LIST() (sage.misc.explain_pickler.PickleExplainer method), 330
list() (sage.misc.lazy_list.lazy_list_generic method), 164
list_function() (in module sage.misc.latex), 308
list_function() (in module sage.misc.mathml), 292
list_of_subfunctions() (in module sage.misc.rest_index_of_methods), 439
list_packages() (in module sage.misc.package), 368
listen() (sage.media.wav.Wave method), 116
load() (in module sage.misc.persist), 318
load_arg() (sage.ext.fast_callable.InstructionStream method), 213
load_const() (sage.ext.fast_callable.InstructionStream method), 213
load_sage_element() (in module sage.misc.persist), 319
load_sage_object() (in module sage.misc.persist), 319
load_session() (in module sage.misc.session), 345
load_submodules() (in module sage.misc.dev_tools), 433
loadable_module_extension() (in module sage.misc.sageinspect), 417
loads() (in module sage.misc.persist), 320
loads() (sage.misc.persist.SageUnpickler class method), 317
log() (in module sage.misc.functional), 15
log() (sage.ext.fast_eval.FastDoubleFunc method), 223
lognormvariate() (in module sage.misc.prandom), 43
LONG() (sage.misc.explain_pickler.PickleExplainer method), 330
LONG1() (sage.misc.explain_pickler.PickleExplainer method), 330
LONG4() (sage.misc.explain_pickler.PickleExplainer method), 330
LONG_BINPUT() (sage.misc.explain_pickler.PickleExplainer method), 330
LONG_BINGET() (sage.misc.explain_pickler.PickleExplainer method), 330
LONG_INPUT() (sage.misc.explain_pickler.PickleExplainer method), 330
long_seed() (sage.misc.randstate.randstate method), 38
LookupNameMaker (class in sage.misc.pickle), 233
Lrs (class in sage.features.lrs), 384

**M**
MainClass (class in sage.misc.nested_class), 343
MainClass.NestedClass (class in sage.misc.nested_class), 343
MainClass.NestedClass.NestedSubClass (class in sage.misc.nested_class), 343
make_index() (sage.databases.sql_db.SQLDatabase method), 103
make_None() (in module sage.misc.persist), 321
make_primary_key() (sage.databases.sql_db.SQLDatabase method), 104
make_unique() (sage.databases.sql_db.SQLDatabase method), 104
map_threaded() (in module sage.misc.map_threaded), 79
MARK() (sage.misc.explain_pickle.PickleExplainer method), 330
math_parse() (in module sage.misc.html), 291
MathJax (class in sage.misc.html), 290
MathJaxExpr (class in sage.misc.html), 291
MathML (class in sage.misc.mathml), 292
mathml() (in module sage.misc.mathml), 292
matrix_column_alignment() (sage.misc.latex.Latex method), 298
matrix_delimiters() (sage.misc.latex.Latex method), 298
max_height (sage.ext.fast_eval.FastDoubleFunc attribute), 224
MethodDecorator (class in sage.misc.method_decorator), 65
Mime (class in sage.misc.graphics_file), 114
mime() (sage.structure.graphics_file.GraphicsFile method), 113
minimal_polynomial() (in module sage.misc.functional), 15
minpoly() (in module sage.misc.functional), 16
modify_for_nested_pickle() (in module sage.misc.nested_class), 341
module
  sage.arith.srange, 80
  sage.databases.sql_db, 92
  sage.docs.instancedoc, 407
  sage.ext.fast_callable, 202
  sage.ext.fast_eval, 219
  sage.ext.memory, 441
  sage.features, 374
  sage.features.bliss, 380
  sage.features.csdp, 381
  sage.features.databases, 381
  sage.features.fes, 382
  sage.features.gap, 382
  sage.features.graph_generators, 383
  sage.features.lrs, 384
  sage.media.wav, 115
  sage.misc.abstract_method, 49
  sage.misc.banner, 349
  sage.misc.benchmark, 397
  sage.misc.bindable_class, 52
  sage.misc.c3, 441
  sage.misc.c3_controlled, 443
  sage.misc.cachefunc, 166
  sage.misc.call, 68
  sage.misc.callable_dict, 68
  sage.misc.citation, 384
  sage.misc.classcall_metaclass, 61
  sage.misc.classgraph, 427
  sage.misc.constant_function, 60
  sage.misc.conveting_dict, 69
  sage.misc.copying, 385
  sage.misc.cython, 229
  sage.misc.decorators, 54
  sage.misc.defaults, 1
  sage.misc.dev_tools, 428
  sage.misc.dist, 373
  sage.misc.edit_module, 424
  sage.misc.element_with_label, 83
  sage.misc.explain_pickle, 327
  sage.misc.fast_methods, 65
  sage.misc.flatten, 72
  sage.misc.fpickle, 326
  sage.misc.func_persist, 315
  sage.misc.function_mangling, 434
  sage.misc.functional, 2
  sage.misc.getusage, 426
  sage.misc.gperftools, 404
  sage.misc.html, 288
  sage.misc.inherit_comparison, 63
  sage.misc.inline_fortran, 232
  sage.misc.latex, 293
  sage.misc.latex_macros, 312
  sage.misc.lazy_attribute, 144
  sage.misc.lazy_format, 154
  sage.misc.lazy_import, 155
  sage.misc.lazy_import_cache, 159
  sage.misc.lazy_list, 159
  sage.misc.lazy_string, 165
  sage.misc.map_threaded, 79
  sage.misc.mathml, 292
  sage.misc.messaging, 89
  sage.misc.method_decorator, 65
  sage.misc.misc, 122
  sage.misc.misc_c, 135
  sage.misc.mrange, 73
  sage.misc.multireplace, 79
  sage.misc.nested_class, 339
  sage.misc.nested_class_test, 397
  sage.misc.object_multiplexer, 65
  sage.misc.package, 366
  sage.misc.parser, 356
  sage.misc.parser, 233
  sage.misc.persist, 315
  sage.misc.prandom, 41
  sage.misc.profiler, 403
  sage.misc.python, 242
  sage.misc.random_testing, 394
  sage.misc.randstate, 30
  sage.misc.remote_file, 88
  sage.misc.repr, 261
  sage.misc.reset, 351
  sage.misc.rest_index_of_methods, 437
  sage.misc.sage_eval, 243
  sage.misc.sage_input, 263
  sage.misc.sage_ostools, 90
INDEX
save_cache_file() (in module sage.misc.lazy_import), 158
save_session() (in module sage.misce.session), 345
search() (in module sage.misce.search), 73
search_def() (in module sage.misc.sagedoc), 360
search_doc() (in module sage.misce.sagedoc), 361
search_src() (in module sage.misce.sagedoc), 361
sec() (sage.ext.fast_eval.FastDoubleFunc method), 224
seed (in module sage.misce.randstate), 40
seed() (sage.misce.randstate.randstate method), 39
series_precision() (in module sage.misce.defaults), 1
set_cache() (sage.misc.cachefunc.CachedFunction method), 176
set_cache() (sage.misc.cachefunc.CachedMethodCallerNoArgs method), 181
set_default_variable_name() (in module sage.misce.defaults), 1
set_edit_template() (in module sage.misce.edit_module), 425
set_editor() (in module sage.misce.edit_module), 426
set_names() (sage.misc.parser.LookupNameMaker method), 233
set_random_seed() (in module sage.misce.randstate), 41
set_seed_gap() (sage.misce.randstate.randstate method), 39
set_seed_gp() (sage.misce.randstate.randstate method), 39
set_seed_libc() (sage.misce.randstate.randstate method), 40
set_seed_ntl() (sage.misce.randstate.randstate method), 40
set_seed_pari() (sage.misce.randstate.randstate method), 40
set_series_precision() (in module sage.misce.defaults), 1
set_state() (in module sage.misce.stopgap), 118
set_values() (sage.media.wav.Wave method), 117
set_verbose() (in module sage.misce.verbose), 143
set_verbose_files() (in module sage.misce.verbose), 144
setdefault() (sage.misc.converting_dict.KeyConvertingDict method), 71
setdefault() (sage.misc.weak_dict.WeakValueDictionary method), 200
SETITEM() (sage.misc.explain_pickle.PickleExplainer method), 331
SETITEMS() (sage.misc.explain_pickle.PickleExplainer method), 331
Sh (class in sage.misce.sh), 246
share() (sage.misc.explain_pickle.PickleExplainer method), 333
share() (sage.misce.sage_input.SageInputBuilder method), 277

SIE_assign (class in sage.misce.misce.input), 181
SIE_binary (class in sage.misce.misce.input), 266
SIE_call (class in sage.misce.misce.input), 266
SIE_dict (class in sage.misce.misce.input), 266
SIE_gen (class in sage.misce.misce.input), 267
SIE_getattr (class in sage.misce.misce.input), 267
SIE_import_name (class in sage.misce.misce.input), 267
SIE literal (class in sage.misce.misce.input), 268
SIE literal_stringrep (class in sage.misce.misce.input), 267
SIE_subscript (class in sage.misce.misce.input), 268
SIE_tuple (class in sage.misce.misce.input), 269
SIE Unary (class in sage.misce.misce.input), 269
sin() (sage.ext.fast_eval.FastDoubleFunc method), 225
Singleton (class in sage.misce.misce.functional), 225
sized_iter (class in sage.misce.misce.functional), 141
skip_TESTS_block() (in module sage.misce.misce.functional), 364
slice_seconds() (sage.media.wav.Wave method), 117
slice_unpickle() (in module sage.misce.misce.functional), 117
some_elements() (sage.misc.sage_unittest.InstanceTester method), 388
some_tuples() (in module sage.misce.misce.functional), 132
source_fd (sage.misc.sage_ostools.redirection attribute), 91
source_file (sage.misc.sage_ostools.redirection attribute), 91
specialize (class in sage.misce.misce.functional), 59
sphinxify() (in module sage.misce.misce.functional), 366
spkg_installation_hint() (sage.features.PackagesSystem method), 378
split() (sage.typeset.character_art.CharacterArt method), 251
SQLException (class in sage.misce.misce.functional), 93
SQLDatabase (class in sage.misce.misce.functional), 107
sqrt() (sage.ext.fast_eval.FastDoubleFunc method), 225
squarefree_part() (in module sage.misce.misce.functional), 26
srange() (in module sage.misce.misce.functional), 80
standard_packages() (in module sage.misce.misce.functional), 372
start() (sage.misc.gperftools.Profiler method), 405
StaticFile (class in sage.misce.misce.functional), 379

Index 477
STOP() (sage.misc.explain_pickle.PickleExplainer method), 331
stop() (sage.misc.gperftools.Profiler method), 406
stopgap() (in module sage.misc.stopgap), 118
StopgapWarning, 118
str_function() (in module sage.misc.latex), 309
str_function() (in module sage.misc.mathml), 292
STRING() (sage.misc.explain_pickle.PickleExplainer method), 331
strunc() (in module sage.misc.misc), 132
suboptions (in module sage.misc.decorators), 59
subsets() (in module sage.misc.misc), 132
sum() (sage.misc.sage_input.SageInputBuilder method), 277
symbolic_prod() (in module sage.misc.functional), 26
symbolic_sum() (in module sage.misc.functional), 27

t
T
table (class in sage.misc.table), 283
tan() (sage.ext.fast_eval.FastDoubleFunc method), 225
tanh() (sage.ext.fast_eval.FastDoubleFunc method), 225
template_fields() (in module sage.misc.edit_module), 426
test() (sage.misc.parser.Tokenizer method), 241
test_add_commutes() (in module sage.misc.random_testing), 395
test_add_is_mul() (in module sage.misc.random_testing), 395
test_fake_startup() (in module sage.misc.lazy_import), 158
test_pickle() (in module sage.misc.explain_pickle), 337
TestAppendList (class in sage.misc.explain_pickle), 334
TestAppendNonlist (class in sage.misc.explain_pickle), 335
TestBuild (class in sage.misc.explain_pickle), 335
TestBuildSetstate (class in sage.misc.explain_pickle), 335
TestGlobalFunnyName (class in sage.misc.explain_pickle), 335
TestGlobalNewName (class in sage.misc.explain_pickle), 335
TestGlobalOldName (class in sage.misc.explain_pickle), 335
TestReduceGetinitargs (class in sage.misc.explain_pickle), 336
TestReduceNoGetinitargs (class in sage.misc.explain_pickle), 336
testSuite (class in sage.misc.sage_unittest), 390
testSuiteFailure, 394
timeCall() (in module sage.misc.classcall_metaclasse), 62
tmp_dir() (in module sage.misc.sage_input.SageInputBuilder method), 87
tmp_filename() (in module sage.misc.sage_input.SageInputBuilder method), 87
token_to_str() (in module sage.misc.parser), 242
tokeneater() (sage.misc.sageinspect.BlockFinder method), 411
Tokenizer (class in sage.misc.parser), 238
top() (sage.misc.gperftools.Profiler method), 406
transpose() (in module sage.misc.functional), 30
transpose() (in module sage.misc.table.table method), 288
TrivialClasscallMetaClass (class in sage.features), 380
TrivialUniqueRepresentation (class in sage.features), 380
try_read() (in module sage.misc.misc), 333
TUPLE() (sage.misc.explain_pickle.PickleExplainer method), 331
TUPLE1() (sage.misc.explain_pickle.PickleExplainer method), 331
TUPLE2() (sage.misc.explain_pickle.PickleExplainer method), 331
TUPLE3() (sage.misc.explain_pickle.PickleExplainer method), 331
tuple_function() (in module sage.misc.latex), 310
tuple_function() (in module sage.misc.mathml), 292
typecall() (in module sage.misc.classcall_metaclasse), 62

U
UNICODE() (sage.misc.explain_pickle.PickleExplainer method), 331
unicode_art() (in module sage.typeset.unicode_art), 259
unicode_subscript() (in module sage.typeset.unicode_art), 260
unicode_superscript() (in module sage.typeset.unicode_art), 260
UnicodeArt (class in sage.typeset.unicode_art), 259
uniform() (in module sage.misc.prandom), 46
union() (in module sage.misc.misc), 134
union() (in module sage.databases.sql_db.SQLQuery method), 110
uniq() (in module sage.misc.misc), 135
Unknown (class in sage.misc.misc), 48
UnknownClass (class in sage.misc.misc), 48
UnknownError, 48
unpickle_all() (in module sage.misc.persist), 325
unpickle_append() (in module sage.misc.explain_pickle), 338
unpickle_build() (in module sage.misc.explain_pickle), 338
unpickle_extension() (in module sage.misc.explain_pickle), 338
unpickle_function() (in module sage.misc.fpickle), 327

478
unpickle_global() (in module sage.misc.persist), 325
unpickle_instantiate() (in module sage.misc.explain_pickle), 338
unpickle_newobj() (in module sage.misc.explain_pickle), 339
unpickle_persistent() (in module sage.misc.explain_pickle), 339
unpickleMethod() (in module sage.misc.fpickle), 327
unpickleModule() (in module sage.misc.fpickle), 327
unset_verbose_files() (in module sage.misc.verbose), 144
update() (sage.misc.converting_dict.KeyConvertingDict method), 71
use_variable() (sage.misc.sage_input.SageInputBuilder method), 278

V
vacuum() (sage.databases.sql_db.SQLDatabase method), 106
validate() (sage.structure.graphics_file.Mime class method), 114
value() (sage.ext.fast_callable.ExpressionConstant method), 208
values() (sage.media.wav.Wave method), 117
values() (sage.misc.cachefunc.FileCache method), 185
values() (sage.misc.sage.input.SageInput method), 201
var() (sage.ext.fast_callable.ExpressionTreeBuilder method), 210
variable_index() (sage.ext.fast_callable.ExpressionVariable method), 211
variable_names() (in module sage.misc.defaults), 2
vector() (sage.media.wav.Wave method), 117
vector_delimiters() (sage.misc.latex.Latex method), 299
verbose() (in module sage.misc.verbose), 144
verify_column() (in module sage.databases.sql_db), 112
verify_operator() (in module sage.databases.sql_db), 112
verify_same() (in module sage.misc.sage_input), 281
verify_si_answer() (in module sage.misc.sage_input), 282
verify_type() (in module sage.databases.sql_db), 113
version() (in module sage.misc.banner), 350
version_dict() (in module sage.misc.banner), 350
view() (in module sage.misc.latex), 310
Viewer (class in sage.misc.viewer), 353
virtual_memory_limit() (in module sage.misc.getusage), 427
visit_BoolOp() (sage.misc.sageinspect.SageArgSpecVisitor method), 412
visit_Compare() (sage.misc.sageinspect.SageArgSpecVisitor method), 412
visit_Dict() (sage.misc.sageinspect.SageArgSpecVisitor method), 413
visit_List() (sage.misc.sageinspect.SageArgSpecVisitor method), 413
visit_Name() (sage.misc.sageinspect.SageArgSpecVisitor method), 413
visit_NameConstant() (sage.misc.sageinspect.SageArgSpecVisitor method), 414
visit_Num() (sage.misc.sageinspect.SageArgSpecVisitor method), 414
visit_Str() (sage.misc.sageinspect.SageArgSpecVisitor method), 415
visit_Tuple() (sage.misc.sageinspect.SageArgSpecVisitor method), 415
visit_UnaryOp() (sage.misc.sageinspect.SageArgSpecVisitor method), 415
vonmisesvariate() (in module sage.misc.prandom), 46

W
walltime() (in module sage.misc.misc), 135
warning() (in module sage.misc.superseded), 121
Wave (class in sage.media.wav), 115
weak_cached_function() (in module sage.misc.cachefunc), 191
WeakCachedFunction (class in sage.misc.cachefunc), 191
WeakValueDictEraser (class in sage.misc.sageinspect), 195
WeakValueDictionary (class in sage.misc.sageinspect), 196
weibullvariate() (in module sage.misc.prandom), 46
width() (sage.typeset.character_art.CharacterArt method), 251
WithEqualityById (class in sage.misc.sageinspect), 251

X
xinterval() (in module sage.misc.functional), 30
xmrange (class in sage.misc.mrange), 76
xmrange_iter (class in sage.misc.mrange), 77
xsrange() (in module sage.arith.srange), 82

Z
ZZ_seed() (sage.misc.randstate.randstate method), 38

Index