Sage 9.5 Reference Manual: Utilities

Release 9.5

The Sage Development Team

Jan 31, 2022
1 General Infrastructure 1
2 Programming Utilities 53
3 Code Evaluation 251
4 Formatted Output 269
5 Saving and Loading Sage Objects 337
6 Interactive Usage Support 369
7 Development Tools 397
8 Low-Level Utilities 451
9 Indices and Tables 469

Python Module Index 471
Index 473
1.1 Default Settings

AUTHORS: William Stein and David Kohel

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

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

INPUT:

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

EXAMPLES:

```python
sage: from sage.misc.defaults import latex_variable_names
sage: latex_variable_names(0)
()  
sage: latex_variable_names(1, 'a')
('a',)
```
```
sage: latex_variable_names(2, 'beta')
('beta_0', 'beta_1')
sage: latex_variable_names(2, r'\beta')
('\beta_0', '\beta_1')
```

`sage.misc.defaults.series_precision()`

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

EXAMPLES:

```python
sage: series_precision()
20
```

`sage.misc.defaults.set_default_variable_name(name, separator='')`

Change the default variable name and separator.

`sage.misc.defaults.set_series_precision(prec)`

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

EXAMPLES:

```python
sage: set_series_precision(5)
sage: series_precision()
```
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:

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

Vectors and matrices may also have their entries approximated:

```python
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
```

(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.additive_order(x)`

Return the additive order of \( x \).

**EXAMPLES:**

```

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

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

Return the base field over which \( x \) is defined.

**EXAMPLES:**

```

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

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

```

sage: R.base_field()
Traceback (most recent call last):
...
AttributeError: 'PolynomialRing_dense_mod_p_with_category' object has no attribute 'base_field'
```

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

Return the base ring over which \( x \) is defined.

**EXAMPLES:**

```

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

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

Return the fixed basis of \( x \).

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

sage.misc.functional.category(x)
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: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
(continues on next page)
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)
<class 'sage.rings.integer.Integer'>
sage: type(coerce(QQ,5))
<class 'sage.rings.rational.Rational'>

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

EXAMPLES:

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

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

EXAMPLES:

```sage
M = matrix([[2, 3], [3, 4]])
M.decomposition()

G.<a,b> = DirichletGroup(20)
c = a*b
d = c.decomposition(); d

d[0].parent()

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

EXAMPLES:
```
```sage
denominator(17/11111)
11111
denominator(R.<x> = PolynomialRing(QQ))
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)
A = M([[1,2,3,4,5,6,7,8,9]])
det(A)

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

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

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

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

(continues on next page)
sage.misc.functional.discriminant(x)
Return the discriminant of x.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^{29} - 17*x - 1, 'alpha')
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

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

EXAMPLES:

```python
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:

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

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

EXAMPLES:

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

1.2. Functional notation
sage: gen(GF(7))
1
sage: A = AbelianGroup(1, [23])

f

sage.misc.functional.gens(x)
Return the generators of x.

EXAMPLES:

sage: R.<x,y> = SR[

sage: gens(R)
(x, y)

sage: A = AbelianGroup(5, [5,5,7,8,9])

sage: gens(A)
(f0, f1, f2, f3, f4)

sage.misc.functional.hecke_operator(x, n)
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:
```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
sage: integral(sin(x),x)
-cos(x)
```

```python
sage: y = var('y')
sage: integral(sin(x),y)
y*sin(x)
sage: integral(sin(x), x, 0, pi/2)
1
sage: sin(x).integral(x, 0,pi/2)
1
```

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

```python
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?
```

```python
sage.misc.functional.integrate(x, *args, **kwds)
Return an indefinite or definite integral of an object x.
```

First call `x.integral()` and if that fails make an object and integrate it using Maxima, maple, etc, as specified by algorithm.
For symbolic expression calls `sage.calculus.calculus.integral()` - see this function for available options.

**EXAMPLES:**

```python
sage: f = cyclotomic_polynomial(10)
sage: integral(f)
1/5*x^5 - 1/4*x^4 + 1/3*x^3 - 1/2*x^2 + x
```

```python
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))...
sage: 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:**
sage: R = PolynomialRing(QQ, 'x')
sage: is_commutative(R)
true

doctest:...DeprecationWarning: use X.is_commutative() or X in Rings().Commutative()
See https://trac.sagemath.org/32347 for details.
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)
doctest:...DeprecationWarning: use X.is_field() or X in Fields()
See https://trac.sagemath.org/32347 for details.
True

sage.misc.functional.is_integrally_closed(x)

Return whether \( x \) is integrally closed.

EXAMPLES:

sage: is_integrally_closed(QQ)
doctest:...DeprecationWarning: use X.is_integrally_closed()  
See https://trac.sagemath.org/32347 for details.
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)
(continues on next page)
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()
   2

   We lift an element of a quotient polynomial ring:

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

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

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

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

   EXAMPLES:

   sage: log(e^2)
   2

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

   sage: log(1000,10)
   3

   The synonym \ln\ can only take one argument:

   sage: ln(RDF(10))
   2.302585092994046
   sage: ln(2.718)
   0.999896315728952
   sage: ln(2.0)
   0.693147180559945
sage: ln(float(-1))
3.141592653589793j
sage: ln(complex(-1))
3.141592653589793j

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

sage: log(1024, 2)
10
sage: RDF(log(1024, 2))
10.0
sage: log(10, 4)
1/2*log(10)/log(2)

The log function works for negative numbers, complex numbers, and symbolic numbers too, picking the branch with angle between $-\pi$ and $\pi$:

sage: log(-1+0*I)
I*pi
sage: log(CC(-1))
3.14159265358979*I
sage: log(-1.0)
3.14159265358979*I

Small integer powers are factored out immediately:

sage: log(4)
2*log(2)

The hold parameter can be used to prevent automatic evaluation:

sage: log(-1,hold=True)
log(-1)

(continues on next page)
For input zero, the following behavior occurs:

```
sage: log(0)
-Infinity
sage: log(CC(0))
-infinity
sage: log(0.0)
-inf
```

The log function also works in finite fields as long as the argument lies in the multiplicative group generated by the base:

```
sage: F = GF(13); g = F.multiplicative_generator(); g
2
sage: a = F(8)
sage: log(a,g); g^log(a,g)
3
8
Traceback (most recent call last):
  ... ValueError: no logarithm of 8 found to base 3 modulo 13
sage: log(F(9), 3)
2
```

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

```
sage: R = Zp(5); R
5-adic Ring with capped relative precision 20
sage: a = R(16); a
1 + 3*5 + O(5^20)
sage: log(a)
3*5 + 3*5^2 + 3*5^4 + 3*5^6 + 3*5^7 + 2*5^8 + 5^9 + 5^11 + 2*5^12 + 5^13 + 3*5^15 + 2*5^16 + 4*5^17 + 3*5^18 + 3*5^19 + O(5^20)
```

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

EXAMPLES:

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

sage.misc.functional.minpoly(x, var='x')
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:**
```
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
```

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

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

Vectors and matrices may also have their entries approximated:

```
sage: v = vector(RDF, [1, 2, 3])
sage: v.n()
```

```
sage: v = vector(CDF, [1, 2, 3])
sage: v.n()
```

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

```
sage: A = matrix(QQ, 2, 3, range(6))
sage: A.n()
```

(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:

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

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

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

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

```
sage: N(pi, digits=1)
3.2
sage: N(pi, digits=2)
3.1
sage: N(100*pi, digits=1)
320.
sage: N(100*pi, digits=2)
310.
```

In the following example, pi and 3 are both approximated to two bits of precision and then subtracted, which kills two bits of precision:
sage: N(pi, prec=2)
3.0
sage: N(3, prec=2)
3.0
sage: N(pi - 3, prec=2)
0.00

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 \( v = (v_1, v_2, \ldots, v_n) \), also called the Euclidean norm, is defined as

\[ |v| = \sqrt{\sum_{i=1}^{n} |v_i|^2} \]

where \(|v_i|\) is the complex modulus of \( v_i \). The Euclidean norm is often used for determining the distance between two points in two- or three-dimensional space.

For complex numbers, the function returns the field norm. If \( c = a + bi \) is a complex number, then the norm of \( c \) is defined as the product of \( c \) and its complex conjugate:

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

The norm of a complex number is different from its absolute value. The absolute value of a complex number is defined to be the square root of its norm. A typical use of the complex norm is in the integral domain \( \mathbb{Z}[i] \) of Gaussian integers, where the norm of each Gaussian integer \( c = a + bi \) is defined as its complex norm.

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

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

See also:

* sage.matrix.matrix2.Matrix.norm()
* sage.modules.free_module_element.FreeModuleElement.norm()
* sage.rings.complex_double.ComplexDoubleElement.norm()
* sage.rings.complex_mpfr.ComplexNumber.norm()
EXAMPLES:

The norm of vectors:

```
sage: z = 1 + 2*I
sage: norm(vector([z]))
sqrt(5)
sage: v = vector([-1,2,3])
sage: norm(v)
sqrt(14)
sage: _ = var("a b c d", 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:

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

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

The norm of complex numbers:

```
sage: z = 2 - 3*I
sage: norm(z)
13
sage: a = randint(-10^10, 100^10)
sage: b = randint(-10^10, 100^10)
sage: z = a + b*I
sage: bool(norm(z) == a^2 + b^2)
True
```

The complex norm of symbolic expressions:

```
sage: a, b, c = var("a, b, c")
sage: assume((a, 'real'), (b, 'real'), (c, 'real'))
sage: z = a + b*I
sage: bool(norm(z).simplify() == a^2 + b^2)
True
sage: norm(a + b).simplify()
a^2 + 2*a*b + b^2
sage: v = vector([a, b, c])
sage: bool(norm(v).simplify() == sqrt(a^2 + b^2 + c^2))
(continues on next page)
```
**True**
sage: forget()

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

Return the numerator of `x`.

**EXAMPLES:**

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

```python
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:**

```python
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: numerical_approx(pi^2 + e, digits=50)
12.58788622954840385419778471228813633070946500941
sage: a = CC(-5).n(prec=40)
sage: b = ComplexField(40)(-5)
sage: a == b
True
sage: parent(a) is parent(b)
True
```

(continues on next page)
sage: numerical_approx(9)
9.00000000000000

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

sage: v.n(prec=20)
(1.0000, 2.0000, 3.0000)

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

sage: n(u, digits=5)
(0.50000, 0.33333, 0.25000)

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

sage: A = matrix(QQ, 2, 3, range(6))
sage: A.n()
[0.000000000000000 1.00000000000000 2.00000000000000]
[ 3.00000000000000 4.00000000000000 5.00000000000000]

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

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

sage: x=N(pi, digits=3); x
3.14
Increasing the precision of a floating point number is not allowed:

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

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

```
sage: N(pi, digits=1)
3.2
sage: N(pi, digits=2)
3.1
sage: N(100*pi, digits=1)
320.
sage: N(100*pi, digits=2)
310.
```

In the following example, \(\pi\) and 3 are both approximated to two bits of precision and then subtracted, which kills two bits of precision:

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

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

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

(continues on next page)
sage: x
x

sage.misc.functional.objgens(x)

EXAMPLES:

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

sage.misc.functional.order(x)

Return the order of x.

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

EXAMPLES:

sage: C = CyclicPermutationGroup(10)
sage: order(C)
10
sage: F = GF(7)
sage: order(F)
7

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

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

EXAMPLES:

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

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

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

EXAMPLES:

sage: quotient(5, 6)
5/6
sage: quotient(5., 6.)
0.8333333333333333
sage: R.<x> = ZZ[]; R
Univariate Polynomial Ring in x over Integer Ring
sage: I = Ideal(R, x^2+1)
sage: quotient(R, I)
Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus 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
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
sage: E = EllipticCurve([0,0,1,-1,0])
sage: rank(E)
1
```

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

**EXAMPLES:**

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

This example addresses trac ticket #23502:

---

1.2. Functional notation
sage: n = round(6); type(n)
<class '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.sqrt(x, *args, **kwds)

INPUT:

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

EXAMPLES:

sage: sqrt(-1)
I
sage: sqrt(2)
sqrt(2)
sage: sqrt(2)^2
2
sage: sqrt(4)
2
sage: sqrt(4,all=True)
[2, -2]
sage: sqrt(x^2)
sqrt(x^2)

For a non-symbolic square root, there are a few options. The best is to numerically approximate afterward:

sage: sqrt(2).n()
1.41421356237310
sage: sqrt(2).n(prec=100)
1.4142135623730950488016887242

Or one can input a numerical type.

sage: sqrt(2.) 1.41421356237310
sage: sqrt(2.000000000000000000000000)
1.41421356237309504880169

To prevent automatic evaluation, one can use the hold parameter after coercing to the symbolic ring:
This illustrates that the bug reported in trac ticket #6171 has been fixed:

```python
sage: a = 1.1
sage: a.sqrt(prec=100)  # this is supposed to fail
Traceback (most recent call last):
... TypeError: ...sqrt() got an unexpected keyword argument 'prec'
sage: sqrt(a, prec=100)
1.0488088481701515469914535137
sage: sqrt(4.00, prec=250)
2.0000000000000000000000000000000000000000000000000000000000000000000000000
```

One can use numpy input as well:

```python
sage: import numpy
sage: a = numpy.arange(2,5)
sage: sqrt(a)
array([1.41421356, 1.73205081, 2.])
```

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

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

**EXAMPLES:**

```python
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:

```python
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: 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

```python
```
- '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
```

```
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](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
```

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)
```

(continues on next page)
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:

\[
\text{sage: } \sum(\binom{n}{k}x^k, k, 0, n, \text{algorithm} = \text{'maple'}) \quad \# \text{optional - maple}
\]

\[(x + 1)^n\]

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

\[
\text{sage: } \sum(x, x, 1r, 5r)
\]

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:

\[
\text{sage: } M = \text{MatrixSpace(QQ,3,3)}
\]
\[
\text{sage: } A = M([1,2,3,4,5,6,7,8,9])
\]
\[
\text{sage: } \text{transpose}(A)
\]

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

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

Iterator over the integers between \(a\) and \(b\), inclusive.

EXAMPLES:

\[
\text{sage: } I = \text{xinterval}(2,5); I
\]
\[
\text{range}(2, 6)
\]
\[
\text{sage: } 5 \text{ in } I
\]

True
\[
\text{sage: } 6 \text{ in } I
\]

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.

```python
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 \(\mathbb{Z}, \mathbb{R}\), 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 \(\text{ZZ\_random}\) uses a separate NTL random number generator in the main Sage process. And \(\text{random()}\) is from a Python \(\text{random.Random}\) object.

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

```python
sage: set_random_seed(0)
sage: print(rtest())
(303, -0.266166246380421, 1/6, (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, 0, (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()
12345
sage: print(rtest())
(720, -0.612180244315804, 0, (1,3), [ 1, 0, 1, 1, 0 ], 1911581957, 65175, 0.
˓
\rightarrow 8043027951758298)
sage: initial_seed()
12345
```

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
(969, -0.407373370020575, 6/7*x^2 + 1, (1,2,3)(4,5), 985329107, 21461, 0.
˓
\rightarrow 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
33623774725802488418842839280045662
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()`.

### 1.3. Random Number States
That pretty much covers what you need to know for command-line use of this module. Now let's move to what authors of Sage library code need to know about the module.

### 1.3.2 Library API

First, we'll cover doctesting. Every docstring now has an implicit `set_random_seed(0)` prepended. Any uses of `# random` that are based on random numbers under the control of this module should be removed, and the reproducible answers inserted instead.

This practice has two potential drawbacks. First, it increases the work of maintaining doctests. For instance, in a long docstring that has many doctests that depend on random numbers, a change near the beginning (for instance, adding a new doctest) may invalidate all later doctests in the docstring. To reduce this downside, you may add calls to `set_random_seed(0)` throughout the docstring (in the extreme case, before every doctest).

Second, the `# random` in the doctest served as a signal to the reader of the docstring that the result was unpredictable and that it would not be surprising to get a different result when trying out the examples in the doctest. If a doctest specifically refers to `ZZ.random_element()` (for instance), this is presumably enough of a signal to render this function of `# random` unnecessary. However, some doctests are not obviously (from the name) random, but do depend on random numbers internally, such as the `composition_series` method of a `PermutationGroup`. In these cases, the convention is to insert the following text at the beginning of the `EXAMPLES` section.

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

sage: set_random_seed(0)
```

Note that this call to `set_random_seed(0)` is redundant, since `set_random_seed(0)` is automatically inserted at the beginning of every docstring. However, it makes the example reproducible for somebody who just types the lines from the doctest and doesn’t know about the automatic `set_random_seed(0)`.

Next, let's cover setting the random seed from library code. The first rule is that library code should never call `set_random_seed()`. This function is only for command-line use. Instead, if the library code wants to use a different random seed, it should use `with seed(s)`. This will use the new seed within the scope of the `with` statement, but will revert to the previous seed once the `with` statement is completed. (Or the library can use `with seed()`: to get a seed automatically selected using `os.urandom()` or the current time, in the same way as described for `set_random_seed()` above.)

Ideally, using `with seed(s)`: should not affect the outer random number sequence at all; we will call this property “isolation.” We achieve isolation for most, but not all, of the random number generators in Sage (we fail for generators, such as NTL, that do not provide an API to retrieve the current random number state).

We’ll demonstrate isolation. First, we show the sequence of random numbers that you get without intervening `with seed`.
We get slightly different results with an intervening `with seed`.

```python
sage: set_random_seed(0)
sage: r1 == rtest()
True
sage: with seed(1):
    ....: print(rtest())
    ....: 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: r2m = rtest(); r2m
(443, 0.185001351421963, -2, (1,3), [ 0, 0, 1, 1, 0 ], 53231108, 51295, 0.6170498912488264)
sage: r2m == r2
False
```

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

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

```python
sage: set_random_seed(0)
sage: r1 == rtest()
True
sage: with seed(1):
    ....: print(rtest())
    ....: print(rtest())
(978, 0.0557699430711638, -1/8*x^2 - 1/2*x + 1/2, (1,2,3), [ 1, 0, 0, 0, 1 ], 807447831, ...
    ... 23865, 0.6170498912488264)
(181, 0.607995392046754, -x + 1/2, (2,3)(4,5), [ 1, 0, 0, 1, 1 ], 1010791326, 9693, 0.6170498912488264)
sage: r2m == rtest()
True
```

The NTL results after the `with seed` don't depend on how many NTL random numbers were generated inside the `with seed`.

```python
sage: set_random_seed(0) sage: r1 == rtest() True sage: with seed(1): ....: 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: r2m == rtest() True
```

(In general, the above code is not exactly equivalent to the `with` statement, because if an exception happens in the body, the real `with` statement will pass the exception information as parameters to the `__exit__` method. However, our `__exit__` method ignores the exception information anyway, so the above is equivalent in our case.)
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:

```python
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:

```python
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:

```python
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:

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

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

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

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

EXAMPLES:

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

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

EXAMPLES:

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

c_random()
Returns a 31-bit random number. Intended for internal use only; instead of calling current_randstate().c_random(), it is equivalent (but probably faster) to call the random method of this randstate class.

EXAMPLES:

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

We verify the equivalence mentioned above.

```
sage: from sage.misc.randstate import random
sage: set_random_seed(1207)
sage: random()
2008037228
```

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

EXAMPLES:
sage: set_random_seed(1618)
sage: current_randstate().long_seed()
2560562797745140950807350947089272595

python_random(cls=None, seed=None)

Return a \texttt{random.Random} object. The first time it is called on a given \texttt{randstate}, a new \texttt{random.Random} is created (seeded from the current \texttt{randstate}); the same object is returned on subsequent calls.

It is expected that \texttt{python_random} will only be called on the current \texttt{randstate}.

INPUT:

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

set_seed_gap()

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

EXAMPLES:

sage: set_random_seed(99900000999)
sage: current_randstate().set_seed_gap()
sage: gap.Random(1, 10^50)
14967382633255543447453297768680634540939580077
sage: gap(35).SCRRandomString()
[ 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 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:

```
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:

```
sage: from sage.misc.randstate import _doctest_libc_random
sage: set_random_seed(0xBAD)
sage: current_randstate().set_seed_libc(False)
sage: _doctest_libc_random()  # random
1070075918
```

set_seed_ntl(force)
Checks to see if self was the most recent randstate to seed the NTL random number generator. If not, seeds the generator. If the argument force is True, seeds the generator unconditionally.

EXAMPLES:

```
sage: set_random_seed(2008)
This call is actually redundant; ntl.ZZ_random() will seed the generator itself. However, we put the call in to make the coverage tester happy.
```

```
sage: current_randstate().set_seed_ntl(False)
sage: ntl.ZZ_random(10^40)
149528351177535545945928804789519607
```

set_seed_pari()
Checks to see if self was the most recent randstate to seed the Pari random number generator. If not, seeds the generator.

Note: Since pari 2.4.3, pari’s random number generator has changed a lot. The seed output by getrand() is now a vector of integers.

EXAMPLES:

```
sage: set_random_seed(5551212)
sage: current_randstate().set_seed_pari()
sage: pari.getrand().type()
't_INT'
```
**sage.misc.randstate.seed**

alias of **sage.misc.randstate.randstate**

**sage.misc.randstate.set_random_seed**(seed=None)

Set the current random number seed from the given seed (which must be coercible to a Python long).

If no seed is given, then a seed is automatically selected using `os.urandom()` if it is available, or the current time otherwise.

Type `sage.misc.randstate?` for much more information on random numbers in Sage.

This function is only intended for command line use. Never call this from library code; instead, use with `seed(s)`:

Note that setting the random number seed to 0 is much faster than using any other number.

**EXAMPLES:**

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

### 1.4 Random Numbers with Python API

**AUTHORS:** – Carl Witty (2008-03): new file

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

The functions here are less efficient than the functions in module `random`, because they look up the current random number state on each call.

If you are going to be creating many random numbers in a row, it is better to use the functions in module `sage.misc.randstate` directly.

Here is an example:

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

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

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

```python
sage: with seed(0): test1()
5169
sage: with seed(0): test2()
5169
sage: with seed(1): test1()
5097
sage: with seed(1): test2()
```

(continues on next page)
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)
```python
sage: sample = [expovariate(1000) for i in range(3)]; sample  # random
[0.001206870332283973, 8.340929747302108e-05, 0.00219877067980605]
sage: all(s >= 0.0 for s in sample)
True
```

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

Gamma distribution. Not the gamma function!

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

**EXAMPLES:**

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

**sage.misc.prandom.gauss**(*mu*, *sigma*)

Gaussian distribution.

**EXAMPLES:**

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

**sage.misc.prandom.getrandbits**(*k*)

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

**EXAMPLES:**

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

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

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.

**EXAMPLES:**
sage.misc.prandom.normalvariate(mu, sigma)
Normal distribution.
mu is the mean, and sigma is the standard deviation.

EXAMPLES:

```
sage: [normalvariate(0, 1) for i in range(3)]  # random
[-1.372558980559407, -1.1701670364898928, 0.0432410055110143]
sage: [normalvariate(0, 100) for i in range(3)]  # random
[37.45695875041769, 159.6347743233298, 124.1029321124009]
sage: [normalvariate(1000, 10) for i in range(3)]  # random
[1008.5303090383741, 989.8624892644895, 985.7728921150242]
```

sage.misc.prandom.paretovariate(alpha)
Pareto distribution. alpha is the shape parameter.

EXAMPLES:

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

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, 0, 1, 0, 0, 1, 0, 0, 1, 2, 1, 2, 0, 0, 2, 2]
sage: all(t in [0, 1, 2] for t in s)
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
[0.111439293741037, 0.5143475134191677, 0.04468968524815642, 0.332490606442413]
sage: all(0.0 <= s <= 1.0 for s in sample)
True
```

sage.misc.prandom.randrange(start, stop=None, step=1)
Choose a random item from range(start, stop[, step]).
This fixes the problem with randint() which includes the endpoint; in Python this is usually not what you want.

EXAMPLES:
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
[357009970, 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 + 0.48565248658083227*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
[0.49069775546342537, 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={}".format(n))
    ....:     elif res is False:
    ....:         print("n={}".format(n))
    ....:     else:
    ....:         print("n={}".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)
except UnknownError:
    print("n={} is neither positive nor negative".format(n))

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

AUTHORS:

• Florent Hivert (2010): initial version.
• Ralf Stephan, Vincent Delecroix (2018-2020): redesign

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, or, and to Unknown might fail.

exception sage.misc.unknown.UnknownError
    Bases: TypeError

    Raised whenever Unknown is used in a boolean operation.

    EXAMPLES:

    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

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
- \texttt{optional} – a boolean; defaults to False

The decorator \texttt{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 \texttt{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 \texttt{:meth:`my_method`} computes my_method
    ....:        '''
    ....:        EXAMPLES::
    ....:            ...
    ....:        pass

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

The current policy is that a \texttt{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 = \texttt{True})
    ....:    def my_method(self):
    ....:        '''
    ....:        The method \texttt{:meth:`my_method`} computes my_method
    ....:        '''
    ....:        EXAMPLES::
    ....:            ...
    ....:        pass

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

(continues on next page)
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)
<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:

```python
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:

```python
sage: from sage.misc.nested_class import NestedClassMetaclasse
sage: class Outer(metaclass=NestedClassMetaclasse):
    ....:     class Inner:
    ....:         def __init__(self, *args):
    ....:             print(args)
    ....:         def f(self, *args):
    ....:             print("{} {}").format(self, args)
    ....:         @staticmethod
    ....:         def f_static(*args):
    ....:             print(args)

sage: outer = Outer()
```

By default, when Inner is a class nested in Outer, accessing outer.Inner returns the Inner class as is:
In particular, `outer` is completely ignored in the following call:

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

This is similar to what happens with a static method:

```python
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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

---

### 2.1. Special Base Classes, Decorators, etc.
sage: type(outer.Inner).mro()
[<class 'sage.misc.bindable_class.BoundClass'>,
 <... 'functools.partial'>,
 <... 'object'>]

Still, documentation works as usual:

sage: outer.Inner.__doc__
' some documentation '

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.

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
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
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(1)
1

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

**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:

```python
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:

```python
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:

```python
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(**renames)

**Bases:** object

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

**INPUT:**

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

**EXAMPLES:**
To deprecate an old keyword:

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

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

Decorator factory which should be used in decorators for making sure that meta-information on the decorated callables are retained through the decorator, such that the introspection functions of sage.misc.sageinspect retrieve 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
sage: sage_getsource(g)
'@square...def g(x)...'
```

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

```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
```

(continues on next page)

(continued from previous page)

```python
sage: @diff_arg_dec
def g(x):
    return x
sage: g(1)
3
g(1, some_def_arg=4)
5
```

```python
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
P.<x,y> = QQ[
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
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
def square_for_met(f):
    @sage_wraps(f)
def new_f(self, x):
        return f(self,x)*f(self,x)
    return new_f
class T:
    @square_for_met
def g(self, x):
        "My little method"
        return x
t = T()
sage: t.g(2)
4
t.g.__doc__
'My little method'
```
The bug described in trac ticket #11734 is fixed:

```python
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)
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:

```python
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:
    sage: f = ConstantFunction(3)
    sage: f
    The constant function (...) -> 3
    sage: f()  # doctest: +ELLIPSIS
    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.

```python
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 \texttt{cls} to implement analogues of those special methods for the operations on the class itself.

Currently, the following special methods are supported:

- \texttt{\_\_classcall\_\_} (and \texttt{\_\_classcall\_private\_\_}) for customizing \texttt{cls(...)} (analogue of \texttt{\_\_call\_\_}).
- \texttt{\_\_classcontains\_\_} for customizing membership testing \texttt{x in cls} (analogue of \texttt{\_\_contains\_\_}).
- \texttt{\_\_classget\_\_} for customizing the binding behavior in \texttt{foo.cls} (analogue of \texttt{\_\_get\_\_}).

See the documentation of \texttt{\_\_call\_\_()} and of \texttt{\_\_get\_\_()} and \texttt{\_\_contains\_\_()} for the description of the respective protocols.

\textbf{Warning:} For technical reasons, \texttt{\_\_classcall\_\_}, \texttt{\_\_classcall\_private\_\_}, \texttt{\_\_classcontains\_\_}, and \texttt{\_\_classget\_\_} must be defined as \texttt{staticmethod()}’s, even though they receive the class itself as their first argument.

\textbf{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 \texttt{\_\_set\_\_classcall\_\()\.

\textit{ClasscallMetaclas}s is an extension of the base type.

\textbf{Todo:} find a good name for this metaclass.

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

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

\texttt{sage.misc.classcall\_metaclass.typecall}(\texttt{cls}, *args, **kwds)

Object construction

This is a faster equivalent to \texttt{type.__call__(cls, <some arguments>)}.

INPUT:

- \texttt{cls} – the class used for constructing the instance. It must be a builtin type or a new style class (inheriting from \texttt{object}).

EXAMPLES:

```
sage: from sage.misc.classcall\_metaclass import typecall
sage: class Foo(object): pass
sage: typecall(Foo)
```

(continues on next page)
Warning: `typecall()` doesn't work for old style class (not inheriting from `object`):

```python
sage: class Bar: pass
sage: 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
sage: from sage.misc.classcall_metaclass import (  
    ...: ClasscallMetaclass, CRef, C2, C3, C2C, timeCall)
```

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

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

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

sage: class PRef(object):  
    ...:     def __init__(self, i):  
    ...:         self.i = i+i1
```

For a Python class, compared to the reference class there is a 10% overhead in using `ClasscallMetaClass` if there is no classcall defined:

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

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

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:

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

sage: %timeit timeCall(C2, 1000, i3)  # not tested
625 loops, best of 3: 298 µs per loop
```
Let’s now compare when there is a classcall defined:

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

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

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

```python
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

```python
class sage.misc.inherit_comparison.InheritComparisonClasscallMetaclass
    Bases: sage.misc.classcall_metaclass.ClasscallMetaclass, sage.misc.inherit_comparison.InheritComparisonMetaclass

Combine ClasscallMetaclass with InheritComparisonMetaclass.

class sage.misc.inherit_comparison.InheritComparisonMetaclass
    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 InheritComparisonMetaclass
...:
...: 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__(self):
...:         from sage.misc.inherit_comparison import InheritComparisonMetaclass
...:         return InheritComparisonMetaclass
...:     def __hash__(self):
...:         return 1

''
)sage: a = Derived()
sage: a == a
True
sage: b = DerivedWithRichcmp()
sage: b == b
Calling Base.__richcmp__
True
```

2.1.7 Base Class to Support Method Decorators

AUTHOR:

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

class sage.misc.method_decorator.MethodDecorator(f)
   Bases: sage.structure.sage_object.SageObject

EXAMPLES:

```python
sage: from sage.misc.method_decorator import MethodDecorator
sage: class Foo:
...:     @MethodDecorator
...:     def bar(self, x):
...:         return x**2
sage: f = Foo()
sage: f.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

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

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:

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

(continues on next page)
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:

```python
sage: class D(C):
    ....: pass
sage: import __main__  # This is only needed ...
sage: __main__.C = C  # ... in doctests
sage: __main__.D = D  # same here, only in doctests
sage: orig = type(c)
sage: 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
```
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
MyNonUniqueParent(1)
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`

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

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

**INPUT:**

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

**EXAMPLES:**

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

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

Call the method `name` on `obj`.

This has to exist somewhere in Python!!!
See also:

operator.methodcaller() attrcal()

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

```python
class sage.misc.callable_dict.CallableDict
Bases: dict
Callable dictionary.

This is a trivial subclass of `dict` with an alternative view as a function.

Typical use cases involve passing a dictionary `d` down to some tool that takes a function as input. The usual idiom in such use cases is to pass the `d.__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:

```
sage: from sage.misc.callable_dict import CallableDict
sage: d = CallableDict({'one': 1, 'zwei': 2, 'trois': 3})
sage: d['zwei']
2
sage: d('zwei')
2
```

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

```
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:
• Martin von Gagern (2015-01-31): initial version

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

```
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:

```
sage: K.<x,y> = QQ[]
sage: I = ideal([x^2+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
```

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:

```
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
doctest:warning...:
DeprecationWarning: use 'key in dictionary' syntax instead
See https://trac.sagemath.org/25281 for details.
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
sage: d.pop("3")
Traceback (most recent call last):
... KeyError: ...
```

`setdefault(key, default=None)`

Create a given mapping unless there already exists a mapping for that key.

INPUT:

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

EXAMPLES:

```python
sage: from sage.misc.converting_dict import KeyConvertingDict
sage: d = KeyConvertingDict(int)
sage: d.setdefault("3")
sage: 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:**

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

```python
sage: flatten([[1,1],[1],2])
[1, 1, 1, 2]
sage: flatten([[1,2,3], (4,5), [[1],[2]]])
[1, 2, 3, 4, 5, 1, 2]
sage: flatten([[1,2,3], (4,5), [[1],[2]]],max_level=1)
[1, 2, 3, 4, 5, [1], [2]]
sage: flatten([[[3],[[]]],max_level=0)
[[[3], []]]
sage: flatten([[[3],[[]]],max_level=1)
[[3], []]
sage: flatten([[[3],[[]]],max_level=2]
[3]
```
In the following example, the vector is not flattened because it is not given in the `ltypes` input.

```python
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:

```python
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.

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

Degenerate cases:

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

### 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, 6)
sage: search([1, 4, 6, 7, 8], 5)
(False, 5)
```
2.2.5 Multidimensional enumeration

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

```python
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:

```python
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),
 (continues on next page)
```
sage.misc.mrange.cartesian_product_iterator($X$)

Iterate over the Cartesian product.

**INPUT:**

- $X$ - list or tuple of lists

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

**EXAMPLES:**

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

sage.misc.mrange.mrange($sizes$, $typ$=class 'list')

Return the multirange list with given sizes and type.

This is the list version of $xmlrange$. Use $xmlrange$ 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
sage.misc.mrange.mrange_iter(iter_list, typ=list)

Return the multirange list derived from the given list of iterators.

This is the list version of xmrange_iter(). Use xmrange_iter for the iterator.

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

INPUT:

- iter_list - a finite iterable of finite iterables
- typ - (default: list) a type or class; more generally, something that can be called with a list as input.

OUTPUT: a list

EXAMPLES:

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

Examples that illustrate empty multi-ranges:

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

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

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

AUTHORS:

- Joel B. Mohler

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

Return the multirange iterate with given sizes and type.

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

Use mrange for the non-iterator form.

INPUT:

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

OUTPUT: a generator

EXAMPLES: We create multi-range iterators, print them and also iterate through a tuple version.
We illustrate a few more iterations.

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

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

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

Next we compute the product:

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

Examples that illustrate empty multi-ranges.

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

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

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

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

```
sage: X = ['red', 'apple', 389]
sage: Y = ['orange', 'horse']
sage: for i,j in xmrange([len(X), len(Y))):
    ....:     print((X[i], Y[j]))
('red', 'orange')
('red', 'horse')
('apple', 'orange')
('apple', 'horse')
(389, 'orange')
(389, 'horse')
```
AUTHORS:

- Jon Hanke
- William Stein

class sage.misc.mrange.xmrange_iter(iter_list, typ=<class '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.

```
sage: z = xmrange_iter([[list(range(3))], list(range(2))], tuple); z
xmrange_iter([[0, 1, 2], [0, 1]], <... 'tuple'>)
sage: for a in z:
    ....:     print(a)
(0, 0)
(0, 1)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
```

We illustrate a few more iterations.

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

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

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

Next we compute the product:

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

Examples that illustrate empty multi-ranges.
We use a multi-range iterator to iterate through the Cartesian product of sets.

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'}
```
2.2.7 Threaded map function

sage.misc.map_threaded.map_threaded(function, sequence)
Apply the function to the elements in the sequence by threading recursively through all sub-sequences in the sequence.

EXAMPLES:

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

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

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

AUTHORS:

• William Stein (2007-12); based on feedback from Peter Doyle.

2.2.8 Ranges and the [1,2,...,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,...) 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)
```

(continues on next page)
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,...,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, 12, 14, 16, 18, 20]
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, ..., start+k*step, where start+k*step < end and start+(k+1)*step >= end.
This provides one way to iterate over Sage integers as opposed to Python int's. It also allows you to specify step sizes for such an iteration.

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

`endpoint_tolerance` – used to determine whether or not the endpoint is hit for inexact rings (default 1e-5)

OUTPUT: a list

Note: This function is called srange to distinguish it from the built-in Python range command. The s at the beginning of the name stands for “Sage”.

See also:
xrange() – iterator which is used to implement srange().

EXAMPLES:
```
sage: v = srange(5); v
[0, 1, 2, 3, 4]
sage: type(v[2])
<class 'sage.rings.integer.Integer'>
sage: srange(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: srange(10, 1, -1)
[10, 9, 8, 7, 6, 5, 4, 3, 2]
sage: srange(10, 1, -1, include_endpoint=True)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
sage: srange(1, 10, universe=RDF)
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
sage: srange(1, 10, 1/2)
[1, 3/2, 2, 5/2, 3, 7/2, 4, 9/2, 5, 11/2, 6, 13/2, 7, 15/2, 8, 17/2, 9, 19/2]
sage: srange(1, 5, 0.5)
[1.00000000000000, 1.50000000000000, 2.00000000000000, 2.50000000000000, 3.00000000000000, 3.50000000000000, 4.00000000000000, 4.50000000000000]
sage: srange(0, 1, 0.4)
[0.000000000000000, 0.400000000000000, 0.800000000000000]
sage: srange(1.0, 5.0, include_endpoint=True)
[1.00000000000000, 2.00000000000000, 3.00000000000000, 4.00000000000000, 5.00000000000000]
sage: srange(1.0, 1.1)
[1.00000000000000, 1.10000000000000]
sage: srange(1.0, 1.0)
[]
sage: V = VectorSpace(QQ, 2)
sage: srange(V([0,0]), V([5,5]), step=V([2,2]))
[(0, 0), (2, 2), (4, 4)]
Including the endpoint:
```
sage: srange(0, 10, step=2, include_endpoint=True)
[0, 2, 4, 6, 8, 10]
sage: srange(0, 10, step=3, include_endpoint=True)
[0, 3, 6, 9]
Try some inexact rings:
```
sage: srange(0.5, 1.1, 0.1, universe=RDF, include_endpoint=False)
[0.5, 0.6, 0.7, 0.7999999999999999, 0.8999999999999999, 0.9999999999999999]
sage: srange(0.5, 1, 0.1, universe=RDF, include_endpoint=False)
[0.5, 0.6, 0.7, 0.7999999999999999, 0.8999999999999999]
sage: srange(0.5, 0.9, 0.1, universe=RDF, include_endpoint=False)
[0.5, 0.6, 0.7, 0.7999999999999999]
sage: srange(0, 1.1, 0.1, universe=RDF, include_endpoint=True)
[0.0, 0.1, 0.2, 0.3000000000000004, 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)
[1.00000000000000, 1.50000000000000, 2.00000000000000, 2.50000000000000, 3.00000000000000, 3.50000000000000, 4.00000000000000, 4.50000000000000]
sage: srange(0, 1, 0.4)
[0.000000000000000, 0.400000000000000, 0.800000000000000]
```

Negative steps are also allowed:

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

```
sage.arith.srange.xsrange(start, end=None, step=1, universe=None, coerce=True, include_endpoint=False, endpoint_tolerance=1e-05)
```

Return an iterator over numbers `start, start+step, ..., start+k*step`, where `start+k*step < end` and `start+(k+1)*step >= end`.

This provides one way to iterate over Sage integers as opposed to Python int’s. It also allows you to specify step sizes for such an iteration.

INPUT:

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

OUTPUT: iterator

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

**Warning:** You need to be careful when using this function over inexact rings: the elements are computed via repeated addition rather than multiplication, which may produce slightly different results. For example:

```
sage: sum([1.1] * 10) == 1.1 * 10
False
```

Also, the question of whether the endpoint is hit exactly for a given `start + k*step` is fuzzy for an inexact ring. If `start + k*step = end` for some `k` within `endpoint_tolerance` of being integral, it is
considered an exact hit, thus avoiding spurious values falling just below the endpoint.

EXAMPLES:

```python
generate object at 0x...
g: for i in xsrange(1,5):
....:   print(i)
1
2
3
4
```

See `srange()` for more examples.

### 2.2.9 Elements with labels.

This module implements a simple wrapper class for pairs consisting of an “element” and a “label”. For representation purposes (`repr`, `str`, `latex`), this pair behaves like its label, while the element is “silent”. However, these pairs compare like usual pairs (i.e., both element and label have to be equal for two such pairs to be equal). This is used for visual representations of graphs and posets with vertex labels.

```python
class sage.misc.element_with_label.ElementWithLabel(element, label):
    Bases: object
    Auxiliary class for showing/viewing `Poset`s with non-injective labelings. For hashing and equality testing the resulting object behaves like a tuple `(element, label)`. For any presentation purposes it appears just as `label` would.

    EXAMPLES:
```
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.
- Sebastian Oehms (2021-08-07): add atomic_dir, see trac ticket #32344

class sage.misc.temporary_file.atomic_dir(target_directory)
Bases: object

Write to a given directory using a temporary directory and then rename it to the target directory. This is for creating a directory whose contents are determined uniquely by the directory name. If multiple threads or processes attempt to create it in parallel, then it does not matter which thread created it. Despite this assumption the contents of the directories differ in the examples for demonstration purpose.

See also atomic_write.

INPUT:

- target_directory – the name of the directory to be written. If it exists then the previous contents will be kept.

EXAMPLES:

```python
sage: from sage.misc.temporary_file import atomic_dir
sage: target_dir = tmp_dir()

sage: with atomic_dir(target_dir) as d:
    target_file = os.path.join(d.name, 'test')
    with open(target_file, 'w') as f:
        _ = f.write("First")
        f.flush()

sage: with atomic_dir(target_dir) as e:
    target_file2 = os.path.join(e.name, 'test')
    with open(target_file2, 'w') as g:
        _ = g.write("Second")
        g.flush()

sage: with open(target_file, 'r') as f:
    _ = f.read()
    'First'

sage: with atomic_dir(target_dir) as d:
    target_file = os.path.join(d.name, 'test')
    with open(target_file, 'w') as f:
        _ = f.write("Third")

sage: target = os.path.join(target_dir, 'test')

sage: with open(target, 'r') as h:
    _ = h.read()
    'Second'
```

class sage.misc.temporary_file.atomic_write(target_filename, append=False, mode=438, binary=None, **kwargs)
Bases: object
Write to a given file using a temporary file and then rename it to the target file. This renaming should be atomic on modern operating systems. Therefore, this class can be used to avoid race conditions when a file might be read while it is being written. It also avoids having partially written files due to exceptions or crashes.

This is to be used in a with statement, where a temporary file is created when entering the with and is moved in place of the target file when exiting the with (if no exceptions occurred).

INPUT:

- **target_filename** – the name of the file to be written. Normally, the contents of this file will be overwritten.
- **append** – (boolean, default: False) if True and target_filename is an existing file, then copy the current contents of target_filename to the temporary file when entering the with statement. Otherwise, the temporary file is initially empty.
- **mode** – (default: 0o666) mode bits for the file. The temporary file is created with mode mode & ~umask and the resulting file will also have these permissions (unless the mode bits of the file were changed manually). (Not to be confused with the file opening mode.)
- **binary** – (boolean, default: True on Python 2, False on Python 3) the underlying file is opened in binary mode. If False then it is opened in text mode and an encoding with which to write the file may be supplied.
- ****kwargs – additional keyword arguments passed to the underlying io.open call.

EXAMPLES:

```python
sage: from sage.misc.temporary_file import atomic_write
sage: target_file = tmp_filename()

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

sage: with atomic_write(target_file) as f:
    ....:    _ = f.write("New contents")
    ....:    f.flush()
    ....:    with open(target_file, 'r') as f2:
    ....:        f2.read()  # '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:

```python
sage: from sage.misc.temporary_file import atomic_write
sage: target_file = tmp_filename()

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

sage: with atomic_write(target_file) as f:
    ....:    f.close()
    ....:    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:
with atomic_write(target_file) as f:
  _ = f.write("Newest contents")
raise RuntimeError

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'
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”:
Supposing an encoding means we're writing the file in “text mode” (in the same sense as `io.open`) and so we must write unicode strings:

```python
sage: target_file = tmp_filename()
sage: with atomic_write(target_file, binary=False, encoding='utf-8') as f:
    _ = f.write(u'Hélas')
sage: import io
sage: with io.open(target_file, encoding='utf-8') as f:
    print(f.read())
Hélas
```

Supplying an encoding in binary mode (or other arguments that don’t make sense to `io.open` in binary mode) is an error:

```python
sage: writer = atomic_write(target_file, binary=True, encoding='utf-8')
sage: with writer as f:
    _ = f.write(u'Hello')
Traceback (most recent call last):
...
ValueError: binary mode doesn't take an encoding argument
```

```python
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:
```
```python
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:

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

`sage.misc.temporary_file.tmp_filename(name='tmp_', ext='')`

Create and return a temporary file in $HOME/.sage/temp/hostname/pid/

The temporary file is deleted automatically when Sage exits.

**Warning:** If you need a particular file extension always use `tmp_filename(ext=".foo")`, this will ensure that the file does not yet exist. If you were to use `tmp_filename()+".foo"`, then you might overwrite an existing file!

**INPUT:**

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

**OUTPUT:**

The absolute path of the temporary file created.

**EXAMPLES:**

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

Temporary files are unaccessible by other users:

```python
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:**
```python
sage: url = 'https://www.sagemath.org/files/loadtest.py'
sage: g = get_remote_file(url, verbose=False)  # optional - internet
sage: with open(g) as f: print(f.read())  # optional - internet
    print("hi from the net")
print(2 + 3)
```

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 `$HOME/.sage/init.sage`.

### 2.3.4 Miscellaneous operating system functions

`sage.misc.sage_ostools.have_program(program, path=None)`

Return True if a program executable is found in the path given by path.

**INPUT:**

- **program** - a string, the name of the program to check.
- **path** - string or None. Paths to search for `program`, separated by `os.pathsep`. If None, use the PATH environment variable.

**OUTPUT:** bool

**EXAMPLES:**

```python
sage: from sage.misc.sage_ostools import have_program
sage: have_program('ls')
True
sage: have_program('there_is_not_a_program_with_this_name')
False
sage: 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!")

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()

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

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

hello world!
```

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
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+')
sage: r = redirection(sys.stdout, f, close=False)
sage: with r:
    ....:     print("Line 1")
sage: with r:
    ....:     print("Line 2")
sage: with f:
```

(continues on next page)
The redirection also works for subprocesses:

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

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

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:
Utilities, Release 9.5

- 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:

```python
{'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:

```python
{'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

```python
class sage.databases.sql_db.SQLDatabase(filename=None, read_only=None, skeleton=None)
```

A SQL Database object corresponding to a database file.

INPUT:

- filename -- a string
- skeleton -- a triple-indexed dictionary:

```python
| - 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

(continues on next page)
| `- `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_\rightarrow graphs
```

We will need a table in the database in which to store the graphs, and we specify its structure with a Python dictionary, each of whose keys is the name of a column:

```
sage: from collections import OrderedDict
sage: table_skeleton = OrderedDict([  
....:('graph6', {'sql':'TEXT', 'index':True, 'primary_key':True}),  
....:('vertices', {'sql':'INTEGER'}),  
....:('edges', {'sql':'INTEGER'})  
....: ])  
```

Then we create the table:

```
sage: D.create_table('simon', table_skeleton)  
sage: D.show('simon')  
graph6 vertices edges  
----------------------------------------------  
```

Now that we have the table, we will begin to populate the table with rows. First, add the graph on zero vertices:

```
sage: G = Graph()  
sage: D.add_row('simon',(G.graph6_string(), 0, 0))  
sage: D.show('simon')  
graph6 vertices edges  
----------------------------------------------  
? 0 0  
```

Next, add the graph on one vertex:

```
sage: G.add_vertex()  
0  
sage: D.add_row('simon',(G.graph6_string(), 1, 0))  
sage: D.show('simon')  
```

(continues on next page)
Say we want a database of graphs on four or less vertices:

```sage
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')
```

Additionally, 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
sage: Q = SQLQuery(D, {'table_name':'simon', 'display_cols':['graph6'], 'expression': '{{'vertices','=','4'}})
sage: Q2 = SQLQuery(D, {'table_name':'simon', 'display_cols':['graph6'], 'expression': '{{'edges','=','3'}})
sage: Q = Q.intersect(Q2)
sage: len(Q.query_results())
3
sage: Q.query_results() # random
[('CF', 'CF'), ('CJ', 'CJ'), ('CL', 'CL')]
```

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

Of course, we can save the database to file:
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')
```

<table>
<thead>
<tr>
<th>graph6</th>
<th>vertices</th>
<th>edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>A_</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>BG</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>BW</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Bw</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>C@</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>CB</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>CF</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>CJ</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>CK</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>CL</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>CN</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>C]</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>C^</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>C~</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

```
sage: E.drop_table('simon')
Traceback (most recent call last):
...  
RuntimeError: Cannot drop tables from a read only database.
```

### add_column

The format for this is:

```

```

```
```

INPUT:

- col_dict – a dictionary:
  - key – column name
  - inner key – one of the following:
    - primary_key – boolean, whether column has been set as primary key
    - index – boolean, whether column has been set as index
    - unique – boolean, whether column has been set as unique
    - sql – one of 'TEXT', 'BOOLEAN', 'INTEGER', 'REAL', or other user defined type

EXAMPLES:
```python
sage: from collections import OrderedDict
sage: MonicPolys = SQLDatabase()

sage: MonicPolys.create_table('simon', OrderedDict([('n', {'sql': 'INTEGER', 'index': True}))))

sage: for n in range(20): MonicPolys.add_row('simon', (n,))

sage: MonicPolys.add_column('simon', 'n_squared', {'sql': 'INTEGER', 'index': False}, 0)

sage: MonicPolys.show('simon')

<table>
<thead>
<tr>
<th>n</th>
<th>n_squared</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**add_data** *(table_name, rows, entry_order=None)*

**INPUT:**

- rows – a list of tuples that represent one row of data to add (types should match col types in order)
- entry_order – an ordered list or tuple overrides normal order with user defined order

**EXAMPLES:**

```python
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'])

sage: cur = DB.get_cursor()

sage: (cur.execute('select * from simon')).fetchall()

[(0, 0), (1, 1), (1, 2), (0, 0), (0, 4), (1, 5)]
```

**add_row** *(table_name, values, entry_order=None)*

Add the row described by values to the table table_name. Values should be a tuple, of same length and order as columns in given table.

**NOTE:**

If values is of length one, be sure to specify that it is a tuple of length one, by using a comma, e.g.:
### add_rows

**Syntax:**

```python
def add_rows(table_name, rows, entry_order=None)
```

**Parameters:**

- `table_name`: a string
- `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

**Returns:**

- A list of tuples representing the added rows.

**Examples:**

```python
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'])
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

**Syntax:**

```python
def commit()
```

**Returns:**

- `None`

**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), (0, 0), (0, 4), (1, 5)])
sage: DB.drop_column('simon', 'b2')
sage: DB.commit()
sage: DB.vacuum()
```

### create_table

**Syntax:**

```python
def create_table(table_name, table_skeleton)
```

**Parameters:**

- `table_name`: a string
- `table_skeleton`: a double-indexed dictionary
  - outer key – column name
  - inner key – one of the following:
    - `sql` – column type (e.g., 'INTEGER', 'bool', 'int', etc.)
    - `primary_key` – `True` or `False`
    - `unique` – `True` or `False`
    - `index` – `True` or `False`

**Returns:**

- `None`

**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), (0, 0), (0, 4), (1, 5)])
sage: DB.drop_column('simon', 'b2')
sage: DB.commit()
sage: DB.vacuum()
```
· 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)
sage: D.show('simon')
```

```
graph6 vertices edges
------------------------------------------------------------
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
```
```python
sage: Q = SQLQuery(DB, {
    'table_name': 'lucy',
    'display_cols': ['id', 'a1', 'b2'],
    'expression': ['id', '>=', 3]})
```

```python
sage: DB.delete_rows(Q)
```

```python
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,))
sage: MonicPolys.add_column('simon', 'n_squared', {
    'sql': 'INTEGER'}, 0)
sage: MonicPolys.drop_column('simon', 'n_squared')
sage: MonicPolys.show('simon')
```

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

### drop_data_from_table(table_name)
Remove all rows from table_name.

**EXAMPLES:**

```python
sage: D = SQLDatabase()
sage: D.create_table('simon', {'col1': {'sql': 'INTEGER'}})
```
sage: D.add_row(('simon', (9,)))
sage: D.show('simon')
coll
--------------------
9
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:

```python
sage: MonicPolys = SQLDatabase()
sage: MonicPolys.create_table('simon', {'n': {'sql': 'INTEGER', 'index': True}, 'n2 →': {'sql': 'INTEGER'}})
sage: MonicPolys.drop_index('simon', 'n')
sage: MonicPolys.make_primary_key('simon', 'n2')
sage: MonicPolys.get_skeleton()
{'simon': {'n': {'index': False, 'primary_key': False, 'sql': 'INTEGER', 'unique': False}, 'n2': {'index': False, 'primary_key': False, 'sql': 'INTEGER', 'unique': False}}}
```

**drop_primary_key** *(table_name, col_name)*

Set the column *col_name* in table *table_name* not to be a primary key.

A primary key is something like an index, but its main purpose is to link different tables together. This allows searches to be executed on multiple tables that represent maybe different data about the same objects.

**NOTE:**

This function only changes the column to be non-primary, it does not delete it.

EXAMPLES:

```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.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}}}
```
**drop_table**(table_name)
Delete table table_name from database.

**INPUT:**

- table_name – a string

**EXAMPLES:**

```python
sage: D = SQLDatabase()
sage: D.create_table('simon', {'col1': {'sql': 'INTEGER'}})
sage: D.show('simon')
col1
-------------------
sage: D.drop_table('simon')
sage: D.get_skeleton()
{ }
```

**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}}}
```

**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
```
```python
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')
```

<table>
<thead>
<tr>
<th>n</th>
<th>n2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>81</td>
</tr>
</tbody>
</table>

**get_cursor** *(ignore_warning=None)*

Return a pysqlite cursor for the database connection.

A cursor is an input from which you can execute sqlite commands on the database.

Recommended for more advanced users only.

**EXAMPLES:**

```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:

```python
{'table1':{'coll':{'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
{'aut_grp': {'aut_grp_size': {'index': True,
                             'unique': False,
                             'primary_key': False,
                             'sql': 'INTEGER'},
                ...
               'num_vertices': {'index': True,
                                'unique': False,
                                'primary_key': False,
                                'sql': 'INTEGER'}}
```

### `make_index(col_name, table_name, unique=False)`

Set the column `col_name` in table `table_name` to be an index, that is, a column set up to do quick searches on.

INPUT:

- `col_name` – a string
- `table_name` – a string
- `unique` – requires that there are no multiple entries in the column, makes searching faster

EXAMPLES:

```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': False,
                      '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:
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': True, '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]})
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=?'
```

(continues on next page)
rename_table(table_name, new_name)

Rename the table table_name to new_name.

EXAMPLES:

```sage
D = SQLDatabase()
D.create_table('simon',{'col1':{sql:'INTEGER'}})
D.show('simon')
coll
--------------
D.rename_table('simon', 'lucy')
D.show('simon')
Traceback (most recent call last):
...
RuntimeError: Failure to fetch data.
D.show('lucy')
coll
--------------
```

save(filename)

Save the database to the specified location.

EXAMPLES:

```sage
MonicPolys = SQLDatabase()
MonicPolys.create_table('simon', {'n':{sql:'INTEGER', 'index':True}})
for n in range(20): MonicPolys.add_row('simon', (n,))
tmp = tmp_dir() # replace with your own file path
MonicPolys.save(tmp+'sage.db')
N = SQLDatabase(tmp+'sage.db')
N.show('simon')
n
--------------
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```
show\( (table\_name, **kwds) \)

Show an entire table from the database.

**EXAMPLES:**

```python
sage: DB = SQLDatabase()
sage: DB.create_table('simon', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
sage: DB.add_data('simon', [[0,0], (1,1), (1,2)])
sage: DB.show('simon')
   a1  b2
------------------------
   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()
```

class sage.databases.sql_db.SQLQuery(database, *args, **kwds)

A query for a SQLite database.

**INPUT:**

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

  ```python
  {'table_name': 'tblname', 'display_cols': ['col1', 'col2', 'col3'], 'expression': }[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:**
sage: D = SQLDatabase()
sage: D.create_table('simon',{'a1':{'sql':'bool', 'primary_key':False}, 'b2':{'sql':int, 'primary_key':False}})
sage: D.add_data('simon',[(0,0),(1,2),(2,4)])
sage: r = SQLQuery(D, {'table_name': 'simon', 'display_cols': ['a1'], 'expression': ['b2', '<=', 3]})
sage: r.show()
a1
--------------
0
1

Test that trac ticket #27562 is fixed:

sage: D = SQLDatabase()
sage: D.create_table('simon',{'a1':{'sql':'bool', 'primary_key':False}, 'b2':{'sql':int, 'primary_key':False}})
Traceback (most recent call last):
...
ValueError: Database has no table simon
sage: D.create_table('simon',{'a1':{'sql':'bool', 'primary_key':False}, 'b2':{'sql':int, 'primary_key':False}})
Traceback (most recent call last):
...
ValueError: Database already has a table named simon
sage: SQLQuery(D, {'table_name': 'simon', 'display_cols': ['a1'], 'expression': ['c1', '>' ,2]})
Traceback (most recent call last):
...
ValueError: Table has no column c1

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 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:**

```python
sage: DB = SQLDatabase()
sage: DB.create_table('simon', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
sage: DB.create_table('lucy', {'a1': {'sql': 'bool'}, 'b2': {'sql': 'int'}})
sage: DB.add_data('simon', [(0, 5), (1, 4)])
sage: DB.add_data('lucy', [(1, 1), (1, 4)])
sage: q = SQLQuery(DB, {'table_name': 'lucy', 'display_cols': ['b2'], 'expression': ['a1', '= 1']})
sage: r = SQLQuery(DB, {'table_name': 'simon', 'display_cols': ['a1'], 'expression': ['b2', '<= 6']})
sage: s = q.intersect(r, 'simon', {'lucy': ('a1', 'a1')})
sage: s.get_query_string()
'SELECT lucy.b2, simon.a1 FROM simon INNER JOIN lucy ON simon.a1 = lucy.a1 WHERE ( lucy.a1 = ? ) AND ( simon.b2 <= ? )'
sage: s.query_results()
[(1, 1), (4, 1)]
sage: R = SQLQuery(G, {'table_name': 'graph_data', 'display_cols': ['graph6'], 'expression': ['num_vertices', '= 4']})
```

**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, 'D??', 5, 0), (19, 'D?C', 5, 1), (20, 'D?K', 5, 2), (21, 'D@O', 5, 2), (22, 'D?]', 5, 3)]
sage: R = SQLQuery(G, {'table_name': 'graph_data', 'display_cols': ['graph6'], 'expression': ['num_vertices', '= 4']})
```
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:

  ```python
  {'table_name': {"old_col_name": "new_col_name"}}
  ```

* id_col – reference to a column that can be used as an object identifier for each row

EXAMPLES:

```python
sage: DB = SQLDatabase()
sage: DB.create_table('simon', {'a1':{'sql':'bool', 'primary_key':False}, 'b2':{
         '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
sage: 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')
```

Graph6 Degree Sequence

```
C ?  [0, 0, 0, 0]
C @  [0, 0, 1, 1]
CB   [0, 1, 1, 2]
```
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:

```
{ 'join_table1':('corr_base_col1', 'col1'), 'join_table2':('corr_base_col2', 'col2')}
```

where `join_table1` is to be joined with `join_table2` on `join_table1.col1 = join_table2.col2`

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
```

(continues on next page)
An example skeleton of a database with one table, that table with one column:

```python
{'table1':{'coll':{'primary_key':False, 'unique': True, 'index':True, 'sql':'REAL'}}
```

EXAMPLES:

```python
sage: from sage.databases.sql_db import construct_skeleton
sage: sorted(construct_skeleton(G))
['aut_grp', 'degrees', 'graph_data', 'misc', 'spectrum']
```

**sage.databases.sql db.\texttt{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:**

- \cite{Ha2005}

**EXAMPLES:**

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

**sage.databases.sql db.\texttt{verify_column}(col_dict)**

Verify that \texttt{col_dict} is in proper format, and return a dict with default values filled in. Proper format:

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

**EXAMPLES:**

```python
sage: from sage.databases.sql_db import verify_column
sage: col = {'sql':'BOOLEAN'}
sage: verify_column(col)
{'index': False, 'primary_key': False, 'sql': 'BOOLEAN', 'unique': False}
sage: col = {'primary_key':True, 'sql':'INTEGER'}
sage: verify_column(col)
```
Verify that `operator` is one of the allowed strings. Legal operators include the following strings:

- `='
- `<=`
- `>=`
- `<`
- `>`
- `<>`
- `like`
- `regexp`
- `is null`
- `is not null`

**EXAMPLES:**

```python
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.
```

Verify that the specified `type` is one of the allowed strings.

**EXAMPLES:**

```python
sage: from sage.databases.sql_db import verify_type
sage: verify_type('INT')
True
sage: verify_type('int')
True
sage: verify_type('float')
Traceback (most recent call last):
...  
TypeError: float is not a legal type.
```
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:

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

mim(e()

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:

```python
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:

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

```python
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 wave 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.
This module (and all of sage.media) is deprecated.

**EXAMPLES:**

```python
sage: import sage.media
doctest:warning...
DeprecationWarning: the package sage.media is deprecated
See http://trac.sagemath.org/12673 for details.
```

```python
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)**

    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)

2.6 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:

```python
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:

```python
sage: import sage.misc.stopgap
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:**
This also works for methods:

```
sage: class cls(object):
    ....:     def new_meth(self): return 42
    ....:
    sage: cls().old_meth()  # new_meth
```

Looking at the code:
```
sage: sage.misc.superseded.deprecation(13109, message='the function foo is replaced by bar')
```

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
**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:**

```python
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'}
```

```python
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:**

```python
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.', FutureWarning)
```

```
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 \ b
```

The preparser converts this to multiplications using BackslashOperator().

EXAMPLES:

```
sage: preparse("A \ stem\ matrix(QQ,2,1,[1/3,'2/3'])")
"A * BackslashOperator() * matrix(QQ,Integer(2),Integer(1),[Integer(1)/Integer(3), -'2/3'])"
```
Continued from previous page:

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

```python
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
    2.1083399999999999
    ```

    See also:

    cputime()

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

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

    See also nest()```

2.7. Miscellaneous Useful Functions 127
EXAMPLES:

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

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

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

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

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

**INPUT:**

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

**OUTPUT:**

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

**EXAMPLES:**

```sage
t = cputime()
F = gp.factor(2^199-1)
cputime(t)  # somewhat random
0.01099900000000092
```

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

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

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

(continues on next page)
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)
```

The `sage.misc.misc.forall(S, P)` function returns True and None 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.

```
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)
```

The `sage.misc.misc.get_main_globals()` function returns the main global namespace.

**EXAMPLES:**
This is analogous to \texttt{globals()}, except that it can be called from any function, even if it is in a Python module:

```sage
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 \texttt{\_\_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 \texttt{\_\_getframe\_}.

See \texttt{inject_variable\_test()} for a real test that this works within deeply nested calls in a function defined in a Python module.

\begin{verbatim}
 sage.misc.misc.inject_variable\(\text{name, value, warn=True}\)
 Inject a variable into the main global namespace.

INPUT:

\begin{itemize}
  \item \text{name} – a string
  \item \text{value} – anything
  \item \text{warn} – a boolean (default: \text{False})
\end{itemize}

EXAMPLES:

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

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

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

(continues on next page)
That's because `warn` seems to not reissue twice the same warning:

```python
sage: from warnings import warn
sage: warn("blah")
```

Warnings can be disabled:

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

Use with care!

`sage.misc.misc.inject_variable_test(name, value, depth)`

A function for testing deep calls to `inject_variable`

**EXAMPLES:**

```python
sage: from sage.misc.misc import inject_variable_test
sage: inject_variable_test("a0", 314, 0)
sage: a0
314
sage: inject_variable_test("a1", 314, 1)
sage: a1
314
sage: inject_variable_test("a2", 314, 2)
sage: a2
314
sage: inject_variable_test("a2", 271, 2)
```

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

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

**EXAMPLES:**

```python
sage: from sage.misc.misc import is_in_string
sage: line = "test(#)"
sage: is_in_string(line, line.rfind('#'))
```

`sage.misc.misc.is_iterator(it)`

Tests if it is an iterator.

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

**EXAMPLES:**
sage: it = iter([1,2,3])
sage: is_iterator(it)
True

sage: class wrong():
....:     def __init__(self): self.n = 5
....:     def __next__(self):
....:         self.n -= 1
....:         if self.n == 0: raise StopIteration
....:         return self.n
sage: x = wrong()
sage: is_iterator(x)
False
sage: list(x)
Traceback (most recent call last):
 ...
TypeError: 'wrong' object is not iterable

sage: class good(wrong):
....:     def __iter__(self):
....:         return self
sage: x = good()
sage: is_iterator(x)
True
sage: list(x)
[4, 3, 2, 1]

sage: P = Partitions(3)
sage: is_iterator(P)
False
sage: is_iterator(iter(P))
True

sage.misc.misc.is_sublist(X, Y)
Test whether X is a sublist of Y.

EXAMPLES:

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

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

See also compose() and self-compose()

INPUT:

- \( f \) – a function of one variable

2.7. Miscellaneous Useful Functions
• $n$ – a nonnegative integer
• $x$ – any input for $f$

**OUTPUT:** $f(f(...f(x)...))$, 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(f(f(x))))))))))))

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:**

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

**AUTHORS:**

• David Harvey (2006-09-09)

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

**EXAMPLES:**

```python
sage: pad_zeros(100)
'100'
sage: pad_zeros(10)
'010'
sage: pad_zeros(10, 5)
'00010'
sage: pad_zeros(389, 5)
'00389'
```
sage.misc.misc.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 = []
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:

```python
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:

```python
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:

```python
sage: filename = tmp_filename()
sage: sage_makedirs(filename)
Traceback (most recent call last):
  ...
FileExistsError: [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:

```python
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:

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

```python
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:

```python
sage: import io
sage: filename = tmp_filename()

sage: from sage.misc.misc import try_read
sage: with open(filename, 'w') as fobj:
    ....: _ = fobj.write('a\nb\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('a\nb\nc')

sage: print(try_read(buf))
a
b
c

sage: _ = buf.seek(0); try_read(buf, splitlines=True)
['a\n', 'b\n', 'c']

sage: buf = io.BytesIO(b'a\nb\nc')

sage: try_read(buf) == b'a\nb\nc'
True

sage: _ = buf.seek(0)

sage: try_read(buf, splitlines=True) == [b'a\n', b'b\n', 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:
```
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
doctest:...: DeprecationWarning: sage.misc.misc.union is deprecated...
[1, 2, 3, 4, 5, 6]
sage: union([1,2,3,4,5,6], [5,6]) == answer
True
sage: union((1,2,3,4,5,6), [5,6]) == answer
True
sage: union((1,2,3,4,5,6), set([5,6])) == answer
True
```

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

EXAMPLES:

```python
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
[-13, -5, 1, 3, 8, 13]
```

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

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

OUTPUT:
• float - time in seconds

EXAMPLES:

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

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

2.7. Miscellaneous Useful Functions 139
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 (2008-03-26): Balanced product tree for generators and iterators
- Stefan van Zwam (2013-06-06): Added bitset tests, some docstring cleanup

```python
class sage.misc.misc_c.NonAssociative(left, right=None):
    Bases: object

    This class is to test the balance nature of prod.

    EXAMPLES:

    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))
```

```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:
AUTHORS:

- Joel B. Mohler (2007-10-03): Reimplemented in Cython and optimized

```
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
```
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((-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]
```

(continues on next page)
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(None,4,None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(None,4,-2); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(None,4,4); 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(-2,4,None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(-2,4,4); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(-2,4,4); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(-2,4,None); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(-2,4,4); normalize_index(s,5)==list(range(5))[s]
True
sage: s=slice(-2,-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(-2,4,4); 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(-2,4,4); 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(-2,4,4); normalize_index(s,5)==list(range(5))[s]
True

(continues on next page)
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.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 (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("abcdef")
['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]
sage: running_total([])
[]

class sage.misc.misc_c.sized_iter

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 len(iterable) will be used.

If the iterable does not have the given length, a ValueError is raised during iteration.

EXAMPLES:

sage: from sage.misc.misc_c import sized_iter
sage: list(sized_iter(range(4)))
[0, 1, 2, 3]
sage: list(sized_iter(range(4), 4))
[0, 1, 2, 3]
sage: list(sized_iter(range(5), 4))
Traceback (most recent call last):
  ... ValueError: sequence too long (expected length 4, got more)

sage: list(sized_iter(range(3), 4))
Traceback (most recent call last):
  ... ValueError: sequence too short (expected length 4, got 3)

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

sage: it = sized_iter(range(5), 2)
sage: next(it)
0
sage: next(it)
Traceback (most recent call last):
... ValueError: sequence too long (expected length 2, got more)

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

```
sage: list(sized_iter([], 0))
[]
sage: sized_iter([1], 0)
Traceback (most recent call last):
... ValueError: sequence too long (expected length 0, got more)
```

If no length is given, the iterable must implement `__len__`:

```
sage: sized_iter(x for x in range(4))
Traceback (most recent call last):
... TypeError: object of type 'generator' has no len()
```

### 2.7.3 Verbosity System and Logging in SageMath

#### Howto: Logging

**Using Python's Logging Module**

Import it:

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

Setting the level:

```
sage: logging.getLogger().setLevel(logging.INFO)
```

Log something:

```
sage: logger = logging.getLogger(__name__)
sage: logger.info('Hello. I am talking to you.')
INFO:__main__:Hello. I am talking to you.
```

If we haven’t set the logging level to `logging.INFO`, then the previous wouldn’t have been shown.

```
sage: logger.debug('Hello. I am really talking a lot.')
```

The latter is not shown as the current logging level is only `logging.INFO` and not `logging.DEBUG`.

Reset the level:

```
sage: logging.getLogger().setLevel(logging.WARNING)
```

Warnings are still shown at this default level (`logging.WARNING`):
sage: logger.warning('Hello. I am warning you.')
WARNING:__main__:Hello. I am warning you.

And that's all.

There are a lot more features, see Logging facility for Python.

**Using SageMath’s Verbosity System**

Alternatively, this module provides `verbose()`, `setVerbose()`, `getVerbose()` which can be used as follows:

```python
sage: from sage.misc.verbose import verbose, setVerbose, getVerbose
sage: setVerbose(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.getVerbose()`

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: getVerbose()
0
sage: setVerbose(2)
sage: getVerbose()
2
sage: setVerbose(0)
```

2.7. Miscellaneous Useful Functions
sage.misc.verbose.get_verbose_files()

sage.misc.verbose.set_verbose(level, files='all')
Set the global Sage verbosity level.

   INPUT:
   • level – an integer between 0 and 2, inclusive.
   • files (default: ‘all’): list of files to make verbose, or ‘all’ to make ALL files verbose (the default).

OUTPUT: changes the state of the verbosity flag and possibly appends to the list of files that are verbose.
EXAMPLES:

```
sage: set_verbose(2)
sage: verbose("This is Sage.", level=1)  # not tested
VERBOSE1 (?): This is Sage.
sage: verbose("This is Sage.", level=2)  # not tested
VERBOSE2 (?): This is Sage.
sage: verbose("This is Sage.", level=3)  # not tested
[no output]
sage: set_verbose(0)
```

sage.misc.verbose.set_verbose_files(file_name)
sage.misc.verbose.unset_verbose_files(file_name)

sage.misc.verbose.verbose(mesg="/quotesingle.ts1/quotesingle.ts1", 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:

```
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()
sage: a.x
calculating x in A
3
sage: a.x
3
```

Implementation details

We redo the same example, but opening the hood to see what happens to the internal dictionary of the object:

```python
sage: a = A()
sage: a.__dict__
{'a': 2}
sage: a.x
calculating x in A
3
```

(continues on next page)
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:

\[
\begin{align*}
\text{sage: } & \quad \text{a = A()} \\
\text{sage: } & \quad \text{a.x = 4} \\
\text{sage: } & \quad \text{a.x} \\
\text{sage: } & \quad \text{a.__dict__} \\
\end{align*}
\]

Class binding results in the lazy attribute itself:

\[
\begin{align*}
\text{sage: } & \quad \text{A.x} \\
\end{align*}
\]

\[
\text{<sage.misc.lazy_attribute.lazy_attribute object at ...>}
\]

**Conditional definitions**

The function calculating the attribute may return NotImplemented to declare that, after all, it is not able to do it. In that case, the attribute lookup proceeds in the super class hierarchy:

\[
\begin{align*}
\text{sage: } & \quad \text{class B(A):} \\
& \quad \text{......: } \quad \text{@lazy_attribute} \\
& \quad \text{......: } \quad \text{def x(self):} \\
& \quad \text{......: } \quad \quad \text{if hasattr(self, "y"):} \\
& \quad \text{......: } \quad \quad \quad \text{print("calculating x from y in B")} \\
& \quad \text{......: } \quad \quad \text{return self.y} \\
& \quad \text{......: } \quad \quad \text{else:} \\
& \quad \text{......: } \quad \quad \quad \text{print("y not there; B does not define x")} \\
& \quad \text{......: } \quad \quad \text{return NotImplemented} \\
& \end{align*}
\]

\[
\begin{align*}
\text{sage: } & \quad \text{b = B()} \\
\text{sage: } & \quad \text{b.x} \\
\text{sage: } & \quad \text{y not there; B does not define x} \\
\text{sage: } & \quad \text{calculating x in A} \\
\text{sage: } & \quad \text{3} \\
\text{sage: } & \quad \text{b = B()} \\
\text{sage: } & \quad \text{b.y = 1} \\
\text{sage: } & \quad \text{b.x} \\
\text{sage: } & \quad \text{calculating x from y in B} \\
\text{sage: } & \quad \text{1}
\end{align*}
\]
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:

```python
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:

```python
sage: class B(A):
    ....:     @lazy_attribute
    ....:     def x(self, existence_only=False):
    ....:         if hasattr(self, "y"):
    ....:             if existence_only:
    ....:                 print("testing for x existence in B")
    ....:                 return True
    ....:             else:
    ....:                 print("calculating x from y in B")
    ....:                 return self.y
    ....:         else:
    ....:             print("y not there; B does not define x")
    ....:             return NotImplemented
sage: b = B()
sage: hasattr(b, "x") # todo: not implemented
y not there; B does not define x
sage: b.x
```

(continues on next page)
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
```

(continues on next page)
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'}
```
.. code::

   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:

.. code-block::

   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:

.. code-block::

   sage: cython_code = ["from sage.structure.parent cimport Parent",
   ....:               "from sage.structure.element cimport Element",
   ....:               "cdef class MyElement(Element): pass",
   ....:               "cdef class MyParent(Parent):",
   ....:               "    Element = MyElement"]
   sage: cython('
\n'.join(cython_code))
   sage: P = MyParent(category=Rings())
   sage: P.element_class  # indirect doctest
   <class '...MyElement'>
About descriptor specifications

The specifications of descriptors (see 3.4.2.3 Invoking Descriptors in the Python reference manual) are incomplete w.r.t. inheritance, and maybe even ill-implemented. We illustrate this on a simple class hierarchy, with an instrumented descriptor:

```
sage: class descriptor(object):
    ....:     def __get__(self, obj, cls):
    ....:         print(cls)
    ....:         return 1
sage: class A(object):
    ....:     x = descriptor()
sage: class B(A):
    ....:         pass
```

This is fine:

```
sage: A.x
<__main__.A>
1
```

The behaviour for the following case is not specified (see Instance Binding) when x is not in the dictionary of B but in that of some super category:

```
sage: B().x
<__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
<__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
```

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

    def unimplemented_AB(self):
        return NotImplemented
    @lazy_attribute
    def unimplemented_B_implemented_A(self):
        return NotImplemented

```
import sage

class C(sage.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'
```

(continues on next page)
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
sage: Cl.x
1
```

As for a any 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:

```python
sage: class Cl1(object):
    ....: @lazy_class_attribute
    ....: def x(cls):
    ....:     print("computing x")
    ....:     return 1
sage: Cl1().x
computing x
1
sage: Cl1().x
1
```

**Warning:** The behavior of lazy class attributes with respect to inheritance is not specified. It currently depends on the evaluation order:

```python
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:

```python
sage: QQ._tester().assertTrue(True,
.....:   "%s doesn't contain 0"%IDontLikeBeingPrinted())
```

This behavior can induce major performance penalties when testing. Note that this issue does not impact the usual assert:

```python
sage: assert True, "%s is wrong"%IDontLikeBeingPrinted()
```

We now check that `LazyFormat` indeed solves the assertion problem:

```python
sage: QQ._tester().assertTrue(True,
.....:   LazyFormat("%s is wrong")%IDontLikeBeingPrinted())
sage: QQ._tester().assertTrue(False,
.....:   LazyFormat("%s is wrong")%IDontLikeBeingPrinted())
```

**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:**
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
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
doctest:warning...
UserWarning: Option `at_startup=True` for lazy import ZZ not needed anymore
Integer Ring
```

See also:

`lazy_import()`, `LazyImport`

AUTHOR:

• Robert Bradshaw

```python
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.ensure_startup_finished()
Make sure that the startup phase is finished.

In contrast to `finish_startup()`, this function can be called repeatedly.

sage.misc.lazy_import.finish_startup()
Finish the startup phase.

This function must be called exactly once at the end of the Sage import process (all).
```
sage.misc.lazy_import.get_star_imports(module_name)
Lookup the list of names in a module that would be imported with “import *” either via a cache or actually importing.

EXAMPLES:

```python
sage: from sage.misc.lazy_import import get_star_imports
sage: 'get_star_imports' in get_star_imports('sage.misc.lazy_import')
True
sage: 'EllipticCurve' in get_star_imports('sage.schemes.all')
True
```

sage.misc.lazy_import.is_duringStartup()
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:

```python
sage: lazy_import('sage.rings.all', 'ZZ')
sage: type(ZZ)
<class '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
```

(continues on next page)
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 \texttt{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'])
<class '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:

sage: lazy_import('sage.all', 'Qp', 'my_Qp', deprecation=14275)
sage: my_Qp(5)
doctest:...: DeprecationWarning: Importing my_Qp from here is deprecated. If you need to use it, please import it directly from sage.all
See http://trac.sagemath.org/14275 for details.
5-adic Field with capped relative precision 20

An example of deprecation with a message:

sage: lazy_import('sage.all', 'Qp', 'my_Qp_msg', deprecation=(14275, "This is an example."))
sage: my_Qp_msg(5)
doctest:...: DeprecationWarning: This is an example.
See http://trac.sagemath.org/14275 for details.
5-adic Field with capped relative precision 20

An example of an import relying on a feature:

sage: from sage.features import PythonModule
sage: lazy_import('ppl', 'equation', feature=PythonModule('ppl', spkg='pplpy'))
sage: equation
<built-in function equation>

sage: lazy_import('PyNormaliz', 'NmzListConeProperties', feature=PythonModule(→ 'PyNormaliz', spkg='pynormaliz') # optional - pynormaliz

sage: NmzListConeProperties # optional - pynormaliz
<built-in function NmzListConeProperties>

sage: lazy_import('foo', 'not_there', feature=PythonModule('foo', spkg='non-existing-package'))

sage: not_there
Failed lazy import:
foo is not available.
Importing not_there failed: No module named 'foo'...
No equivalent system packages for ... are known to Sage...

sage.misc.lazy_import.save_cache_file()
Used to save the cached * import names.

sage.misc.lazy_import.test_fake_startup()
For testing purposes only.

Switch the startup lazy import guard back on.

EXAMPLES:

sage: 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

2.8.4 Lazy import cache

This is a pure Python file with no dependencies so it can be used in setup.py.

sage.misc.lazy_import_cache.get_cache_file()
Return the canonical filename for caching names of lazily imported modules.

EXAMPLES:

sage: from sage.misc.lazy_import_cache import get_cache_file
sage: get_cache_file()
'...-lazy_import_cache.pickle'
sage: get_cache_file().startswith(DOT_SAGE)
True
sage: 'cache' in get_cache_file()
True

It should not matter whether DOT_SAGE ends with a slash:

sage: OLD = DOT_SAGE
sage: sage.misc.lazy_import_cache.DOT_SAGE = '/tmp'
sage: get_cache_file().startswith('/tmp/')
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(i2)]
[-2, -2]
```

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

```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 → 01 and 1 → 10:
from sage.misc.lazy_list import lazy_list_generic

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]

w = MyThueMorseWord()
w.lazy list [0, 1, 1, ...]
all(w[i] == ZZ(i).popcount()%2 for i in range(100))
True
w[:500].list() == w[:1000:2].list()
True

Alternatively, you can create the lazy list from an update function:

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)

w2 = lazy_list(update_function=thue_morse_update)
w2.lazy list [0, 1, 1, ...]
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.

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])
sage: l
lazy list [None, [1], [1, 2], ...]
```

Lazy lists behave like lists except that they are immutable:

```sage
l[3::5]
lazy 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]]
```

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
default_preview = 3
sage: lazy_list_formatter(srange(3, 1000, 5), name='list', preview=default_preview)
'list [3, 8, 13, ...]
sage: from sage.misc.lazy_list import lazy_list
default_opening_delimiter = '('
default_closing_delimiter = ')
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]

# laziness 2.8

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:
from sage.misc.lazy_list import lazy_list_from_function
lazy_list_from_function(euler_phi)

lazy_list_from_function(divisors, [None])

class sage.misc.lazy_list.lazy_list_from_iterator
Bases: sage.misc.lazy_list.lazy_list_generic

Lazy list built from an iterator.

EXAMPLES:

from sage.misc.lazy_list import lazy_list
count

m = lazy_list(count()); m

m2 = lazy_list(count())[8:20551:2]
m2

x = iter(m)
[next(x), next(x), next(x)]

y = iter(m)
[next(y), next(y), next(y)]

[3, 3]

loads(dumps(m))

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 the list of values is shorter a RuntimeError will occur. If no value is added then the lazy list is considered finite.
- cache – an optional list to be used as the cache. Be careful that there is no copy.
- stop – an optional integer to specify the length of this lazy list (otherwise it is considered infinite)

class sage.misc.lazy_list.lazy_list_generic

A lazy list

EXAMPLES:
get(i)
Return the element at position i.

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

EXAMPLES:

```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(4)
1
sage: f.get(10)
'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()
[5, 19, 41, 61, 83, 107, 137, 163, 191, 223, 241, 271, 307, 337, 367, 397, 431, ...
  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]
```

2.8. Laziness
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
'l1'

Note that the function is recomputed each time:

```python
sage: L.append(6)
sage: s
'l2'
```

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:**
```sage
from sage.misc.lazy_string import lazy_string
def f(x):
    return "laziness in +str(x"
```
```
s = lazy_string(f, ZZ); s
"laziness in Integer Ring"
```

Here, we demonstrate that the evaluation is postponed until the value is needed, and that the result is not cached:

```sage
class C:
    def __repr__(self):
        print("determining string representation")
        return "a test"
c = C()
s = lazy_string("this is %s", c)
s
determining string representation
"this is a test"
s == "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:

```sage
@cached_function
def test_pfunc(x):
    ""
    Some documentation
    ""
    return -x
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:

```
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:

```
sage: cython_code = [''
cpdef test_meth(self,x):
    return -x'',
    ''from sage.all import cached_method'',
    ''from sage.structure.parent cimport Parent'',
    ''cdef class MyClass(Parent):'',
    ''    @cached_method'',
    ''    def direct_method(self, x):'',
    ''        "Some doc for direct method"'',
    ''        return 2*x'',
    ''    wrapped_method = cached_method(test_meth,name="wrapped_method")'']
sage: cython(os.linesep.join(cython_code))
sage: O = MyClass()
sage: O.direct_method
Cached version of <method 'direct_method' of '...MyClass' objects>
sage: O.wrapped_method
Cached version of <built-in function test_meth>
sage: O.wrapped_method.__name__
'wrapped_method'
sage: O.wrapped_method(5)
-5
sage: O.wrapped_method(5) is O.wrapped_method(5)
True
sage: O.direct_method(5)
10
sage: O.direct_method(5) is O.direct_method(5)
True
```

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

```python
sage: cython_code = ['from sage.all import cached_method, cached_in_parent_method,
                  ...   _Category, Objects",
                  ....: "class MyCategory(Category):",
                  ....:   @cached_method",
                  ....:   def super_categories(self):
                  ....:     return [Objects()]
                  ....: "class ElementMethods",
                  ....:   @cached_method",
                  ....:   def element_cache_test(self):
                  ....:     return -self",
                  ....: "class ParentMethods",
                  ....:   @cached_method",
                  ....:   def one(self):
                  ....:     return self.element_class(self,1)
                  ....: "class MyBrokenElement(Element):
                  ....:   def __init__(self,P,x):
                  ....:     self.x=x
                  ....:     Element.__init__(self,P)
                  ....:   def __neg__(self):
                  ....:     return MyBrokenElement(self.parent(),-self.x)
                  ....:   def __hash__(self):
                  ....:     return hash(self.x)
                  ....:   cpdef _richcmp_(left, right, int op):
                  ....:     return PyObject_RichCompare(left.x, right.x, op)
                  ....:   def raw_test(self):
```

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.

```python
sage: cython_code = ['from sage.structure.element cimport Element,
                  ....: ElementWithCachedMethod", "from cpython.object cimport PyObject_RichCompare",
                  ....: "cdef class MyBrokenElement(Element):
                  ....:   cdef public object x",
                  ....:   def __init__(self,P,x):
                  ....:     self.x=x
                  ....:     Element.__init__(self,P)
                  ....:   def __neg__(self):
                  ....:     return MyBrokenElement(self.parent(),-self.x)
                  ....:   def __hash__(self):
                  ....:     return hash(self.x)
                  ....:   cpdef _richcmp_(left, right, int op):
                  ....:     return PyObject_RichCompare(left.x, right.x, op)
                  ....:   def raw_test(self):
```

(continues on next page)
" return -self",
"cdef class MyElement(ElementWithCachedMethod):",
" cdef public object x",
" def __init__(self,P,x):
    self.x=x",
" ElementWithCachedMethod.__init__(self,P)",
" def __neg__(self):
    return MyElement(self.parent(),-self.x)",
" def __repr__(self):
    return '<%s>'%self.x",
" def __hash__(self):
    return hash(self.x)",
" cpdef _richcmp_(left, right, int op):
    return PyObject_RichCompare(left.x, right.x, op)",
" def raw_test(self):
    return -self",
"from sage.structure.parent cimport Parent",
"cdef class MyParent(Parent):
    Element = MyElement"

sage: cython('
'.join(cython_code))
sage: P = MyParent(category=C)
sage: ebroken = MyBrokenElement(P,5)
sage: e = MyElement(P,5)

The cached methods inherited by the parent works:

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

The cached methods inherited by MyElement works:

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

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

```
sage: e == ebroken
True
sage: type(e) == type(ebroken)
False
```
However, the cache of the other inherited method breaks, although the method as such works:

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

The cache can be emptied:

```
sage: a = test_pfunc(5)
sage: test_pfunc.clear_cache()  
```

```
sage: a is test_pfunc(5)
False
```

```
sage: a = P.one()
sage: P.one.clear_cache()  
```

```
sage: a is P.one()
False
```

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

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

```
sage: b is ebroken.element_via_parent_test()
False
```

Introspection works:

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

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

```
def direct_method(self, x):
    "Some doc for direct method"
    return 2*x
sage: print(sage_getsource(O.direct_method))
```

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)
<class 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
sage: type(I.groebner_basis)
<class 'sage.misc.cachefunc.CachedMethodCaller'>
```

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

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

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

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

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

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

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

sage: K.<a> = Qq(9)
sage: b = a.add_bigoh(1)
sage: c = a + 3
sage: b
a + O(3)
sage: c
a + 3 + O(3^20)
sage: b == c
True
sage: b == a
True
sage: c == a
False

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

sage: hash(b)
Traceback (most recent call last):
...
TypeError: unhashable type: 'sage.rings.padics.qadic_flint_CR.qAdicCappedRelativeElement'
sage: @cached_method
....: def f(x): return x == a
sage: f(b)
True
sage: f(c)  # if b and c were hashable, this would return True
False
sage: b._cache_key()
(..., ((0, 1),), 0, 1)
sage: c._cache_key()
(..., ((0, 1), (1,),), 0, 20)

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

An implementation must make sure that for elements \( a \) and \( b \), if \( a \neq b \), then also \( a._\text{cache}_\text{key}() \neq b._\text{cache}_\text{key}() \). In practice this means that the \_cache_key should always include the parent as its first argument:

```
sage: S.<a> = Qq(4)
sage: d = a.add_bigoh(1)
sage: b._cache_key() == d._cache_key()  # this would be True if the parents were not included
False
```

```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
    • \( \text{name} \) – (optional string) name that the cached version of \( f \) should be provided with
    • \( \text{key} \) – (optional callable) takes the input and returns a key for the cache, typically one would use this to normalize input
    • \( \text{do \_pickle} \) – (optional boolean) whether or not the contents of the cache should be included when pickling this function; the default is not to include them.

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

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

    The inputs to the function must be hashable or they must define \text{sage\_structure\_sage\_object.SageObject\_cache\_key}."

    EXAMPLES:

    ```python
    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:

    ```python
    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")
True
```

### cache

`cached(*args, **kwds)`

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,), ())
sage: f(5)
5
sage: f.cached(5)
5
```

### 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
{}
```

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

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

EXAMPLES:

```python
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:

```python
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:

```python
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:

```python
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,),()]
1

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

EXAMPLES:

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

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.

2.9. Caching
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 \texttt{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 \texttt{sage.structure.sage_object.SageObject\_cache\_key()}.

Examples can be found at \texttt{cachefunc}.

\textbf{class} \texttt{sage.misc.cachefunc.CachedMethod}

\textbf{Bases:} \texttt{object}

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

\textbf{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 \texttt{key} or they must define \texttt{sage.structure.sage_object.SageObject\_cache\_key()}.

EXAMPLES:

\begin{Verbatim}
\textit{sage: class} \texttt{Foo(object):}
\textit{....: @cached\_method}
\textit{....: def} \texttt{f(self, t, x=2):}
\textit{....: \quad print('computing')}
\textit{....: \quad return t**x}
\textit{sage: a = Foo()}
\end{Verbatim}

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

\begin{Verbatim}
\textit{sage: res = a.f(3, 2); res}

\texttt{computing}
\textit{9}
\textit{sage: a.f(t = 3, x = 2) is res}
\texttt{True}
\textit{sage: a.f(3) is res}
\texttt{True}
\end{Verbatim}

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

\begin{Verbatim}
\textit{sage: P.<a,b,c,d> = QQ[]}
\textit{sage: I = P*[a,b]}
\textit{sage: type(I.__class__.gens)}
\texttt{<class 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>}
\end{Verbatim}

So, you would hardly ever see an instance of this class alive.
The parameter `key` can be used to pass a function which creates a custom cache key for inputs. In the following example, this parameter is used to ignore the `algorithm` keyword for caching:

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

The parameter `do_pickle` can be used to enable pickling of the cache. Usually the cache is not stored when pickling:

```python
sage: class A(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:

```python
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)
1
sage: b = loads(dumps(a))
sage: len(b.f.cache)
1
```

Cached methods cannot be copied like usual methods, see trac ticket #12603. Copying them can lead to very surprising results:

```python
sage: class A:
....:     @cached_method
....:     def f(self):
....:         return 1
sage: class B:
....:     g=A.f
....:     def f(self):
....:         return 2
sage: b=B()
```

(continues on next page)
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:

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

cached(*args, **kwds)
Return the result from the cache if available. If the value is not cached, raise KeyError.

EXAMPLES:

```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
```

**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:**

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

**AUTHOR:**

- Simon King (2011-04)

**clear_cache()**
Clear the cache dictionary.

**EXAMPLES:**

---

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

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

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

**is_in_cache()**

Answers whether the return value is already in the cache.

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

**EXAMPLES:**

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

**set_cache(value)**

Override the cache with a specific value.

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

**EXAMPLES:**

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

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

```
sage: I.gens.clear_cache()
sage: I.gens()
[a, b]```
The attempt to assign `None` to the cache fails:

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

```python
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 `do_pickle` is set.)

**EXAMPLES:**

```python
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^3*z - 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:

```python
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:

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

**AUTHOR:**

- Simon King (2011-01)

```python
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 \texttt{\_\_hash\_} would be wrapped by using \texttt{CachedMethod}, then \texttt{hash(c)} will access \texttt{C.\_\_hash\_} and bind it to \texttt{c}, which means that the \texttt{\_\_get\_} method of \texttt{CachedMethod} will be called. But there, we assume that Python has already inspected \texttt{\_\_dict\_}, and thus a \texttt{CachedMethodCaller} will be created over and over again.

Here, the \texttt{\_\_get\_} method will explicitly access the \texttt{\_\_dict\_}, so that \texttt{hash(c)} will rely on a single \texttt{CachedMethodCaller} stored in the \texttt{\_\_dict\_}.

\textbf{EXAMPLES:}

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

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

 sage: hash(c)  # indirect doctest
 compute hash
 5
 sage: hash(c)
 5
\end{verbatim}

\textbf{class} \texttt{sage.misc.cachefunc.DiskCachedFunction}(f, dir, memory_cache=False, key=None)

\textbf{Bases:} \texttt{sage.misc.cachefunc.CachedFunction}

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

\textbf{EXAMPLES:}

\begin{verbatim}
 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
\end{verbatim}

\textbf{class} \texttt{sage.misc.cachefunc.FileCache}(dir, prefix=", memory_cache=False)

\textbf{Bases:} \texttt{object}

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

- \texttt{A} is a tuple of objects \(t\) where each \(t\) is an exact object which is uniquely identified by a short string.

- \texttt{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 \texttt{DiskCachedFunction}. If \texttt{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
gsage: dir = tmp_dir()
gsage: FC1 = FileCache(dir, memory_cache=False, prefix='foo')
gsage: FC2 = FileCache(dir, memory_cache=False, prefix='foo')
gsage: k1 = ((), (('a', 1),))
gsage: t1 = randint(0, 1000)
gsage: k2 = ((), (('b', 1),))
gsage: t2 = randint(0, 1000)
gsage: FC1[k1] = t1
gsage: FC2[k2] = t2
gsage: FC2.clear()
gsage: k1 in FC1 False
gsage: k2 in FC1 False
```

**file_list()**

Return the list of files corresponding to self.

**EXAMPLES:**

```python
sage: from sage.misc.cachefunc import FileCache
gsage: dir = tmp_dir()
gsage: FC = FileCache(dir, memory_cache=True, prefix='t')
gsage: FC[((1,0),)] = 1
gsage: FC[((1,2),(0))] = 2
gsage: FC[((1,),(('a',1),))] = 3
gsage: for f in sorted(FC.file_list()): print(f[len(dir):])
t-.key.sobj
t-.sobj
`.1_2.key.sobj
`.1_2.sobj
`.a-1.1.key.sobj
`.a-1.1.sobj
```

**items()**

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

**EXAMPLES:**

```python
```
```python
sage: from sage.misc.cachefunc import FileCache
sage: dir = tmp_dir()

sage: FC = FileCache(dir, memory_cache = False)

sage: FC[((),())] = 1
sage: FC[((1,2),())] = 2
sage: FC[((1,),(('a',1),))] = 3

sage: I = FC.items()
sage: I.sort(); I
[(((), ()), 1), (((1,), (('a', 1),)), 3), (((1, 2), ()), 2)]
```

**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[((),())] = 1
sage: FC[((1,2),())] = 2
sage: FC[((1,),(('a',1),))] = 3

sage: K = FC.keys()
sage: K.sort(); K
[(((), ()), ((1,), (('a', 1),))), ((1, 2), ())]```

**values()**

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

**EXAMPLES:**

```python
sage: from sage.misc.cachefunc import FileCache
sage: dir = tmp_dir()

sage: FC = FileCache(dir, memory_cache = False)

sage: FC[((),())] = 1
sage: FC[((1,2),())] = 2
sage: FC[((1,),(('a',1),))] = 3

sage: v = FC.values()
sage: v.sort(); v
[1, 2, 3, 4]
```

**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:**

```python
```
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):

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

As an exception meant to improve performance, the most recently computed values are strongly referenced. The
number of strongly cached values can be controlled by the cache keyword.

EXAMPLES:

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

The result is cached:

```python
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:

```python
sage: del a
dsage: del b
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:

```python
sage: @weak CachedFunction(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
```

(continues on next page)
The parameter `key` can be used to ignore parameters for caching. In this example we ignore the parameter `algorithm`:

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

`sage.misc.cachefunc.cache_key(o)`
Helper function to return a hashable key for `o` which can be used for caching.

This function is intended for objects which are not hashable such as $p$-adic numbers. The difference from calling an object's `_cache_key` method directly, is that it also works for tuples and unpacks them recursively (if necessary, i.e., if they are not hashable).

**EXAMPLES:**

```python
sage: from sage.misc.cachefunc import cache_key
sage: K.<u> = Qq(9)
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:

```python
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:

```python
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 \( g = \text{CachedFunction}(f) \) or \( g = \text{cached_function}(f) \) to make a cached version of \( f \), or put `@cached_function` right before the definition of \( f \) (i.e., use Python decorators):

```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:**

```python
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:

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

The user can clear the cache:

```python
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:

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

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

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

```python
sage.misc.cachefunc.cached_in_parent_method(self, inst, *args, **kwds)
A decorator that creates a cached version of an instance method of a class.
```
In contrast to `CachedMethod`, the cache dictionary is an attribute of the parent of the instance to which the method belongs.

ASSUMPTION:
This way of caching works only if

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

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

Examples can be found at `cachefunc`.

### Examples

```python
sage.misc.cachefunc.cached_method(f, name=None, key=None, do_pickle=None)
```
A decorator for cached methods.

**EXAMPLES:**

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

```python
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__)
<class 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
sage: hash(c)
compute hash
5
```

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

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

Different instances have distinct caches:
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:

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

The parameter `do_pickle` can be used if the contents of the cache should be stored in a pickle of the cached method. This can be dangerous with special methods such as `__hash__`:

```python
sage: class C:
    ....:     @cached_method(do_pickle=True)
    ....:     def __hash__(self):
    ....:         return id(self)

sage: import __main__
sage: __main__.C = C
c = C()
hash(c)  # random output
d = loads(dumps(c))
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
c = C()
hash(c)  # random output
d = loads(dumps(c))
hash(d) == hash(c)
False
```

`sage.misc.cachefunc.dict_key(o)`

Return a key to cache object `o` in a dict.

This is different from `cache_key` since the `cache_key` might get confused with the key of a hashable object. Therefore, such keys include `unhashable_key` which acts as a unique marker which is certainly not stored in the dictionary otherwise.
EXAMPLES:

```
sage: from sage.misc.cachefunc import dict_key
dsage: dict_key(42)
42
sage: K.<u> = Qq(9)
dsage: 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:

```
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 \( g = \text{weak_cached_function}(f) \) 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:
However, if there are no strong references left, the result is deleted, and thus a new computation takes place:

```
sage: del a
sage: del b
sage: a = f()
doing a computation
```

Above, we used the cache=0 keyword. With a larger value, the most recently computed values are cached anyway, even if they are not referenced:

```
sage: @weak_cached_function(cache=3)
    def f(x):
        print("doing a computation for x={}").format(x)
        return A()
sage: a = f(1); del a
doing a computation for x=1
sage: a = f(2), f(1); del a
doing a computation for x=2
sage: a = f(3), f(1); del a
doing a computation for x=3
sage: a = f(4), f(1); del a
doing a computation for x=4
doing a computation for x=1
sage: a = f(5), f(1); del a
doing a computation for x=5
```

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

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

### 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
g sage: class Vals(object): pass
g sage: class Keys:
    ....:     def __init__(self, val):
    ....:         self.val = weakref.ref(val)
    ....:     def __hash__(self):
    ....:         return hash(self.val())
    ....:     def __eq__(self, other):
    ....:         return self.val() == other.val()
    ....:     def __ne__(self, other):
    ....:         return self.val() != other.val()

sage: ValList = [Vals() for _ in range(10)]

sage: D = weakref.WeakValueDictionary()
sage: for v in ValList:
    ....:     D[Keys(v)] = v

sage: len(D)
10

sage: del ValList, v

sage: len(D) > 1
True
```

Hence, the defunct items have not been removed from the dictionary.

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

```python
sage: ValList = [Vals() for _ in range(10)]

sage: import sage.misc.weak_dict
sage: D = sage.misc.weak_dict.WeakValueDictionary()
sage: for v in ValList:
    ....:     D[Keys(v)] = v

sage: len(D)
10

sage: del ValList

sage: len(D)
1

sage: del v

sage: len(D)
0
```

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

```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
```

(continues on next page)
With \texttt{WeakValueDictionary}, the behaviour is safer. Note that iteration over a \texttt{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 \texttt{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 \texttt{dict}, in contrast to \texttt{weakref’s} weak value dictionary:

\begin{verbatim}
\end{verbatim}

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

\begin{verbatim}
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:

    \begin{verbatim}
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
    \end{verbatim}

    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:

    \begin{verbatim}
sage: from sage.misc.weak_dict import CachedWeakValueDictionary sage: D = CachedWeakValueDictionary(cache=4)
    \end{verbatim}
\end{verbatim}
```python
class Test(object):
    pass

tmp = Test()
D[0] = tmp
\(\emptyset\) in D
True
del tmp
\(\emptyset\) in D
True
```

```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:
```n```python
from sage.misc.weak_dict import WeakValueDictionary
v = frozenset([1])
D = WeakValueDictionary({1 : v})
len(D)
1
del v
len(D)
0
```

**AUTHOR:**
- Nils Bruin (2013-11)

```python
class sage.misc.weak_dict.WeakValueDictionary
    Bases: dict

    IMPLEMENTATION:

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

    NOTE:

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

    EXAMPLES:
```
sage: import weakref
sage: class Vals(object): pass
sage: class Keys:
    ...:     def __init__(self, val):
    ...:         self.val = weakref.ref(val)
    ...:     def __hash__(self):
    ...:         return hash(self.val())
    ...:     def __eq__(self, other):
    ...:         return self.val() == other.val()
    ...:     def __ne__(self, other):
    ...:         return self.val() != other.val()

sage: ValList = [Vals() for _ in range(10)]
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:

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

items()

The key-value pairs of this dictionary.

EXAMPLES:

sage: import sage.misc.weak_dict
sage: class Vals:
    ...:     def __init__(self, n):
    ...:
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>), ([4], <4>), ([6], <6>), ([7], <7>), ([8], <8>), ([9], <9>)]
```

**iteritems()**

Iterate over the items of this dictionary.

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

**EXAMPLES:**

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

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

**itervalues()**
Iterate over the values of this dictionary.

---

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

**EXAMPLES:**
sage: import sage.misc.weak_dict
sage: class Vals:
....:     def __init__(self, n):
....:         self.n = n
....:     def __repr__(self):
....:         return "<%s>" % self.n
....:     def __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:

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

keys()
The list of keys.

EXAMPLES:

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

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

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

pop(k)
Return the value for a given key, and delete it from the dictionary.

EXAMPLES:

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

**popitem()**

Return and delete some item from the dictionary.

**EXAMPLES:**

```
sage: import sage.misc.weak_dict
tsage: D = sage.misc.weak_dict.WeakValueDictionary()
tsage: D[1] = ZZ
The dictionary only contains a single item, hence, it is clear which one will be returned:

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

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

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

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

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

**EXAMPLES:**

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

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

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

For a non-existing key, the default value is stored and returned:
values()
Return the list of values.

EXAMPLES:

```python
sage: import sage.misc.weak_dict
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()
L = [Vals(n) for n in range(10)]
D = sage.misc.weak_dict.WeakValueDictionary(enumerate(L))
```

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

```python
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:
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:

```plaintext
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)
```

```plaintext
sage: timeit('wilk.subs(x=30)')  # random, long time
625 loops, best of 3: 1.43 ms per loop
```

```plaintext
sage: fc_wilk = fast_callable(wilk, vars=[x])
```

```plaintext
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=`:

```plaintext
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 \( \sin(x) + 3^x \times 2 \) with `domain=D` would be equivalent to \( D(D(\sin(D(x)))) + D(D(3) \times D(D(x)^2))) \). (This example also demonstrates the one exception to the general rule: if an exponent is an integral constant, then it is not wrapped with `D()`.)

At first glance, this seems like a very bad idea if you want to compute quickly. And it is a bad idea, for types where we don’t have a special interpreter. It’s not too bad of a slowdown, though. To mitigate the costs, we check whether the value already has the correct parent before we call `D`.

We don’t yet have a special interpreter with domain `ZZ`, so we can see how that compares to the generic `fc_wilk` example above:

```plaintext
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:

```plaintext
sage: fc_wilk_rdf = fast_callable(wilk, vars=[x], domain=RDF)
```

```plaintext
sage: timeit('fc_wilk_rdf(30.0)')  # random, long time
625 loops, best of 3: 7 us per loop
```

The domain does not need to be a Sage type; for instance, `domain=float` also works. (We actually use the same fast interpreter for `domain=float` and `domain=RDF`; the only difference is that when `domain=RDF` is used, the return value is an RDF element, and when `domain=float` is used, the return value is a Python float.)

```plaintext
sage: fc_wilk_float = fast_callable(wilk, vars=[x], domain=float)
```

```plaintext
sage: timeit('fc_wilk_float(30.0)')  # random, long time
625 loops, best of 3: 5.04 us per loop
```
We also have support for RR:

```sage
sage: fc_wilk_rr = fast_callable(wilk, vars=[x], domain=RR)
sage: timeit('fc_wilk_rr(30.0)')  # random, long time
625 loops, best of 3: 13 us per loop
```

For CC:

```sage
sage: fc_wilk_cc = fast_callable(wilk, vars=[x], domain=CC)
sage: timeit('fc_wilk_cc(30.0)')  # random, long time
625 loops, best of 3: 23 us per loop
```

And support for CDF:

```sage
sage: fc_wilk_cdf = fast_callable(wilk, vars=[x], domain=CDF)
sage: timeit('fc_wilk_cdf(30.0)')  # random, long time
625 loops, best of 3: 10.2 us per loop
```

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).

```sage
sage: K.<x,y,z> = QQ[]
sage: p = 10*y + 100*z + x
sage: fp = fast_callable(p)
sage: fp(1,2,3)
321
```

But you can also specify the variable names to override the default ordering (you can include extra variable names here, too).

```sage
sage: fp = fast_callable(p, vars=('x','w','z','y'))
```

For symbolic expressions, you need to specify the variable names, so that `fast_callable()` knows what order to use.

```sage
sage: var('y,z,x')
(y, z, x)
sage: f = 10*y + 100*z + x
sage: ff = fast_callable(f, vars=(x,y,z))
sage: ff(1,2,3)
321
```

You can also specify extra variable names:

```sage
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 to `fast_callable()` an expression with common subexpressions.

Internally, `fast_callable()` works in two stages: it constructs an expression tree from its argument, and then it builds a fast evaluator from that expression tree. You can bypass the first phase by building your own expression tree and passing that directly to `fast_callable()`, using an `ExpressionTreeBuilder`. 

208 Chapter 2. Programming Utilities
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(v_0, v_1), add(v_1, v_1)), floordiv(v_1, v_1)), add(mul(3.
˓→1415900000000, v_0), mul(1729, v_1))))
```

Many sage/Python built-in functions are specially handled; for instance, when evaluating an expression involving `sin()` over `RDF`, the C math library function `sin()` is called. Arbitrary functions are allowed, but will be much slower since they will call back to Python code on every call; for example, the following will work.

```python
sage: def my_sqrt(x): return pow(x, 0.5)
sage: e = etb.call(my_sqrt, v1); e
{my_sqrt}(add(mul(add(v_0, v_1), add(v_1, v_1)), floordiv(v_1, v_1)))
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:**

```python
sage: var('x')
x
sage: f = fast_callable(sqrt(x^7+1), vars=[x], domain=float)
sage: f(1)
1.414213562373095
sage: f.op_list()
[('load_arg', 0), ('ipow', 7), ('load_const', 1.0), 'add', 'sqrt', 'return']
```
To interpret that last line, we load argument 0 (‘x’ in this case) onto the stack, push the constant 7.0 onto the stack, call the pow function (which takes 2 arguments from the stack), push the constant 1.0, add the top two arguments of the stack, and then call sqrt.

Here we take sin of the first argument and add it to f:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: 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()
[('load_arg', 0), ('ipow', 7), ('load_const', 1), 'add', ('py_call', <function sqrt at .. →>, 1), ('load_arg', 0), ('py_call', sin, 1), 'add', 'return']
```

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 `T 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):
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
class sage.ext.fast_callable.CompilerInstrSpec(n_inputs, n_outputs, parameters)
    Bases: object

    Describe a single instruction to the fast_callable code generator.

    An instruction has a number of stack inputs, a number of stack outputs, and a parameter list describing extra
    arguments that must be passed to the InstructionStream.instr method (that end up as extra words in the code).

    The parameter list is a list of strings. Each string is one of the following:
    • ‘args’ - The instruction argument refers to an input argument of the wrapper class; it is just appended to the
      code.
    • ‘constants’, ‘py_constants’ - The instruction argument is a value; the value is added to the corresponding
      list (if it’s not already there) and the index is appended to the code.
    • ‘n_inputs’, ‘n_outputs’ - The instruction actually takes a variable number of inputs or outputs (the n_inputs
      and n_outputs attributes of this instruction are ignored). The instruction argument specifies the number of
      inputs or outputs (respectively); it is just appended to the code.

class sage.ext.fast_callable.Expression
    Bases: object

    Represent an expression for fast_callable.

    Supports the standard Python arithmetic operators; if arithmetic is attempted between an Expression and a non-
    Expression, the non-Expression is converted to an expression (using the __call__ method of the Expression’s
    ExpressionTreeBuilder).

    EXAMPLES:

    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:

    sage: from sage.ext.fast_callable import ExpressionTreeBuilder
    sage: etb = ExpressionTreeBuilder(vars=(x,))
    sage: x = etb(x)
    sage: abs(x)
    abs(v_0)

(continues on next page)
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: type(etb.call(sin, x))
<class 'sage.ext.fast_callable.ExpressionCall'>
```

arguments()

Return the arguments from this ExpressionCall.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: etb.call(sin, x).arguments()
[v_0]
```

function()

Return the function from this ExpressionCall.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: etb.call(sin, x).function()
sin
```

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.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:
if_false()

Return the false branch of an ExpressionChoice.

EXAMPLES:

```python
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()
v_0
```

if_true()

Return the true branch of an ExpressionChoice.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
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:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: type(etb(3))
<class 'sage.ext.fast_callable.ExpressionConstant'>
```

value()

Return the constant value of an ExpressionConstant.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: etb(3).value()
3
```

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,))
```

```python
type(etb.var('x')^17)
<class 'sage.ext.fast_callable.ExpressionIPow'>
```

### base()
Return the base from this `ExpressionIPow`.

**EXAMPLES:**

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
```

```python
sage: (etb(33)^42).base()
33
```

### exponent()
Return the exponent from this `ExpressionIPow`.

**EXAMPLES:**

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
```

```python
sage: (etb(x)^(-1)).exponent()
-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(vars=(x,))
```

```python
sage: x = etb.var('x')
sage: (x+3)*5
mul(add(v_0, 3), 5)
```

### 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.

**EXAMPLES:**

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
```

```python
sage: etb.call(cos, x)
cos(v_0)
sage: etb.call(sin, 1)
sin(1)
```

(continues on next page)
### choice

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
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:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
sage: type(etb.var(x))
<class 'sage.ext.fast_callable.ExpressionVariable'>
```

variable_index()

Return the variable index of an ExpressionVariable.

EXAMPLES:

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=(x,))
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:

```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('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']
```

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:
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)
<class '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)
sage: instr_stream.has_instr('return')
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.load_arg(5)
sage: instr_stream.current_op_list()
[('load_arg', 5), ('load_arg', 3), ('load_arg', 5)]
```

`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.

```python
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, \( \text{IntegerPowerFunction}(2) \) is the squaring function; \( \text{IntegerPowerFunction}(-1) \) 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 \(a \leq n \leq b\), or True, if ipow should always be used. When ipow cannot be used, then we fall back on calling IntegerPowerFunction.

See the class docstring for CompilerInstrSpec for more information.

NOTE: You must not modify the metadata.

by_opcode
by_opname
ipow_range

class sage.ext.fast_callable.Wrapper
Bases: object

The parent class for all fast_callable wrappers. Implements shared behavior (currently only debugging).

get_orig_args()
Get the original arguments used when initializing this wrapper.

(Probably only useful when writing doctests.)

EXAMPLES:
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: 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: 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:

```python
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
```

We have a special fast interpreter for domain=CDF:

```python
sage: f = fast_callable(expr, vars=('z', 'x', 'y'))
sage: f(1, 2, 3)
sin(2) + 12
sage: K.<x> = QQ[]
sage: 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, 0.5, add, load_arg, 0, mul, load_const, -0.010526315789473684, add, load_arg, 0, mul, load_arg, 0, return]
sage: fp(3.14159)
-552.4182988917153
sage: K.<x,y,z> = QQ[]
sage: p = x*y^2 + 1/3*y^2 - x*z - y*z
sage: fp = fast_callable(p, domain=RDF)
sage: fp.op_list()
[load_const, 0.0, load_const, 1.0, load_arg, 0, ipow, 2, mul, load_arg, 0, add, load_arg, 1, ipow, 2, mul, add, load_arg, 0, ipow, 1, add, load_arg, 0, mul, load_arg, 2, ipow, 1, mul, add, load_arg, 0, 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)
21.831120464939584
```

From Sage, we can create an ExpressionTreeBuilder:

```python
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(ipow(v_0, 2), v_1)
sage: fc = fast_callable(expr, domain=float)
```

(continues on next page)
Check that fast_callable also works for symbolic functions with evaluation functions:

```python
sage: def evalf_func(self, x, y, parent):
    return parent(x*y) if parent is not None
    else x*y
sage: x, y = var('x,y')
```

```python
sage: f = function('f', evalf_func=evalf_func)
```

```python
sage: fc = fast_callable(f(x, y), vars=[x, y])
```

```python
sage: fc(3, 4)
```

And also when there are complex values involved:

```python
sage: def evalf_func(self, x, y, parent):
    return parent(I*x*y) if parent is not None else I*x*y
```

```python
sage: g = function('g', evalf_func=evalf_func)
```

```python
sage: fc = fast_callable(g(x, y), vars=[x, y])
```

```python
sage: fc(3, 4)
```

```python
sage: fc2 = fast_callable(g(x, y), domain=complex, vars=[x, y])
```

```python
sage: fc3 = fast_callable(g(x, y), domain=float, vars=[x, y])
```

```python
sage: fc3(3, 4)
```

```python
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)
<class '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()))
sage: builtins[sin]
'sin'
sage: builtins[ln]
'log'
```

`sage.ext.fast_callable.op_list(args, metadata)`

Given a dictionary with the result of calling `get_current` on an `InstructionStream`, and the corresponding interpreter metadata, return a list of the instructions, in a simple somewhat human-readable format.

For debugging only. (That is, it’s probably not a good idea to try to programmatically manipulate the result of this function; the expected use is just to print the returned list to the screen.)

There’s probably no reason to call this directly; if you have a wrapper object, call `op_list` on it; if you have an `InstructionStream` object, call `current_op_list` on it.

**EXAMPLES:**

```python
sage: from sage.ext.interpreters.wrapper_rdf import metadata
sage: from sage.ext.fast_callable import InstructionStream, op_list
```

(continues on next page)
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.

The solution implemented in this module, by Robert Bradshaw (2008-10), has been superseded by `fast_callable()`. All that remains here is a compatible interface function `fast_float()`.

**AUTHORS:**


```python
sage.ext.fast_eval.fast_float(f, old=None, expect_one_var=False, *vars)
```

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` – deprecated, do not use
- `expect_one_var` – don’t give deprecation warning if `vars` is omitted, as long as expression has only one var

**EXAMPLES:**

```python
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.

```python
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  
```
2.11 Features

2.11.1 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 Feature which can be tested for at runtime. It can be of various kinds, most prominently an Executable in the PATH, a PythonModule, 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()
```

```python
sage: Executable(name="random", executable="randomOochoz6x", spkg="random", url="http://...rand.om").require()  # optional - sage_spkg
Traceback (most recent call last):
... FeatureNotPresentError: random is not available.
Executable 'randomOochoz6x' not found on PATH.
...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)
sage: Executable(name="does-not-exist", executable="does-not-exist-xxxxxxyyyyyy").is_present()
```
FeatureTestResult('does-not-exist', False)

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)
```

class sage.features.Feature(name, spkg=None, url=None, description=None)

Bases: sage.features.TrivialUniqueRepresentation

A feature of the runtime environment

INPUT:
- name – (string) name of the feature; this should be suitable as an optional tag for the Sage doctester, i.e., lowercase alphanumeric with underscores (_) allowed; features that correspond to Python modules/packages may use periods (.)
- spkg – (string) name of the SPKG providing the feature
- description – (string) optional; plain English description of the feature
- url – a URL for the upstream package providing the feature

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
```

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)
```

2.11. Features 227
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)
sage: TestFeature("other").is_present()
FeatureTestResult('other', True)
```

**require()**

Raise a `FeatureNotPresentError` if the feature is not present.

**EXAMPLES:**

```python
sage: from sage.features.gap import GapPackage
sage: GapPackage("ve1EeThu").require()
Traceback (most recent call last):
... FeatureNotPresentError: gap_package_ve1EeThu is not available.
```

**resolution()**

Return a suggestion on how to make `is_present()` pass if it did not pass.

**OUTPUT:**

A string.

**EXAMPLES:**

```python
sage: from sage.features import Executable
sage: Executable(name="CSDP", spkg="csdp", executable="theta", url="https://github.com/dimpase/csdp").resolution()  # optional - sage_spkg
'...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` (*feature, reason=None, resolution=None*)

A missing feature error.

**EXAMPLES:**

```python
sage: from sage.features import Feature, FeatureTestResult
sage: class Missing(Feature):
    ....:     def _is_present(self):
    ....:         return False
```
Missing(name="missing").require()
Traceback (most recent call last):
...
FeatureNotPresentError: missing is not available.

resolution
Initialize self. See help(type(self)) for accurate signature.

class sage.features.FeatureTestResult(feature, is_present, reason=None, resolution=None)
Bases: object
The result of a 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
'retm'

resolution
Initialize self. See help(type(self)) for accurate signature.

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:
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:

```python
sage: from sage.features import StaticFile
sage: from sage.misc.temporary_file import tmp_dir
sage: dir_with_file = tmp_dir()
... file exists in search path
sage: search_path = ( '/foo/bar', dir_with_file ) # file is somewhere in the search path
sage: feature = StaticFile(name="file", filename="file.txt", search_path=search_path)
... file path
sage: feature.absolute_path() == file_path
True
```

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 sage.misc.classcall_metaclass.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 :class:`sage.features.pkg_systems.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')]
```

### 2.11.2 Join features

class sage.features.join_feature.JoinFeature(name, features, spkg=None, url=None, description=None)

Bases: sage.features.Feature

Join of several Feature instances.

EXAMPLES:

```python
sage: from sage.features import Executable
sage: from sage.features.join_feature import JoinFeature
sage: F = JoinFeature("shell-boolean",
.....:     (Executable('shell-true', 'true'),
.....:     Executable('shell-false', 'false')))  
sage: F.is_present()
FeatureTestResult('shell-boolean', True)

sage: F = JoinFeature("asdfghjkl",
.....:     (Executable('shell-true', 'true'),
.....:     Executable('xxyyyy', 'xxyyyy-does-not-exist')))  
sage: F.is_present()
FeatureTestResult('xxyyyy', False)
```

is functional()

Test whether the join feature is functional.

EXAMPLES:

```python
sage: from sage.features.latte import Latte
sage: Latte().is_functional()  # optional - latte_int
FeatureTestResult('latte_int', True)
```
2.11.3 Enumeration of all defined features

`sage.features.all.all_features()`
Return an iterable of all features.

EXAMPLES:
```
sage: from sage.features.all import all_features
sage: sorted(all_features(), key=lambda f: f.name) # random
[...Feature('sage.combinat')...]
```

2.11.4 Features for testing the presence of Python modules in the Sage library

`sage.features.sagemath.all_features()`
Return features corresponding to parts of the Sage library.

These features are named after Python packages/modules (e.g., `sage.symbolic`), not distribution packages (`sagemath-symbolics`).

This design is motivated by a separation of concerns: The author of a module that depends on some functionality provided by a Python module usually already knows the name of the Python module, so we do not want to force the author to also know about the distribution package that provides the Python module.

Instead, we associate distribution packages to Python modules in `sage.features.sagemath` via the `spkg` parameter of `Feature`.

EXAMPLES:
```
sage: from sage.features.sagemath import all_features
sage: list(all_features())
[...Feature('sage.combinat'), ...]
```

class `sage.features.sagemath.sage__combinat`
Bases: `sage.features.join_feature.JoinFeature`
A `Feature` describing the presence of `sage.combinat`.

EXAMPLES:
```
sage: from sage.features.sagemath import sage__combinat
sage: sage__combinat().is_present()  # optional - sage.combinat
FeatureTestResult('sage.combinat', True)
```

class `sage.features.sagemath.sage__geometry__polyhedron`
Bases: `sage.features.PythonModule`
A `Feature` describing the presence of `sage.geometry.polyhedron`.

EXAMPLES:
```
sage: from sage.features.sagemath import sage__geometry__polyhedron
sage: sage__geometry__polyhedron().is_present()  # optional - sage.geometry.polyhedron
FeatureTestResult('sage.geometry.polyhedron', True)
```

class `sage.features.sagemath.sage__graphs`
Bases: `sage.features.join_feature.JoinFeature`
A Feature describing the presence of sage.graphs.

EXAMPLES:
```
sage: from sage.features.sagemath import sage__graphs
sage: sage__graphs().is_present()  # optional - sage.graphs
FeatureTestResult('sage.graphs', True)
```

```python
class sage.features.sagemath.sage_plot
    Bases: sage.features.join_feature.JoinFeature

    A Feature describing the presence of sage.plot.

    EXAMPLES:
    ```python
    sage: from sage.features.sagemath import sage__plot
    sage: sage__plot().is_present()  # optional - sage.plot
    FeatureTestResult('sage.plot', True)
    ```
```

```python
class sage.features.sagemath.sage__rings__number_field
    Bases: sage.features.join_feature.JoinFeature

    A Feature describing the presence of sage.rings.number_field.

    EXAMPLES:
    ```python
    sage: from sage.features.sagemath import sage__rings__number_field
    sage: sage__rings__number_field().is_present()  # optional - sage.rings.number_field
    FeatureTestResult('sage.rings.number_field', True)
    ```
```

```python
class sage.features.sagemath.sage__rings__real_double
    Bases: sage.features.PythonModule

    A Feature describing the presence of sage.rings.real_double.

    EXAMPLES:
    ```python
    sage: from sage.features.sagemath import sage__rings__real_double
    sage: sage__rings__real_double().is_present()  # optional - sage.rings.real_double
    FeatureTestResult('sage.rings.real_double', True)
    ```
```

```python
class sage.features.sagemath.sage__symbolic
    Bases: sage.features.join_feature.JoinFeature

    A Feature describing the presence of sage.symbolic.

    EXAMPLES:
    ```python
    sage: from sage.features.sagemath import sage__symbolic
    sage: sage__symbolic().is_present()  # optional - sage.symbolic
    FeatureTestResult('sage.symbolic', True)
    ```
```

```python
class sage.features.sagemath.sagemath_doc_html
    Bases: sage.features.StaticFile

    A Feature which describes the presence of the documentation of the Sage library in HTML format.

    EXAMPLES:
```

2.11. Features
2.11.5 Features for testing the presence of package systems

class sage.features.pkg_systems.PackageSystem(name, spkg=None, url=None, description=None)
    Bases: sage.features.Feature

    A Feature describing a system package manager.

    EXAMPLES:

    sage: from sage.features.pkg_systems 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.pkg_systems 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.pkg_systems.PipPackageSystem(name, spkg=None, url=None, description=None)
        Bases: sage.features.pkg_systems.PackageSystem

        A Feature describing the Pip package manager.

        EXAMPLES:

        sage: from sage.features.pkg_systems import PipPackageSystem
        sage: PipPackageSystem()  # optional - pip
        Feature('pip')

    class sage.features.pkg_systems.SagePackageSystem(name, spkg=None, url=None, description=None)
        Bases: sage.features.pkg_systems.PackageSystem

        A Feature describing the package manager of the SageMath distribution.

        EXAMPLES:

        sage: from sage.features.pkg_systems import SagePackageSystem
        sage: SagePackageSystem()  # optional - sage-spkg
2.11.6 Features for testing the presence of bliss

class sage.features.bliss.Bliss:
    Bases: sage.features.join_feature.JoinFeature

    A Feature which describes whether the sage.graphs.bliss module is available in this installation of Sage.

    EXAMPLES:

    ::

        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:

    ::

        sage: from sage.features.bliss import BlissLibrary
        sage: BlissLibrary().require() # optional - libbliss

sage.features.bliss.all_features()

2.11.7 Feature for testing the presence of csdp

class sage.features.csdp.CSDP:
    Bases: sage.features.Executable

    A Feature which checks for the theta binary of CSDP.

    EXAMPLES:

    ::

        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.

    EXAMPLES:

    ::

        sage: from sage.features.csdp import CSDP
        sage: CSDP().is_functional() # optional - csdp
        FeatureTestResult('csdp', True)

sage.features.csdp.all_features()
2.11.8 Features for testing the presence of various databases

class sage.features.databases.DatabaseConwayPolynomials
    Bases: sage.features.StaticFile

    A Feature which describes the presence of Frank Luebeck's database of Conway polynomials.

    EXAMPLES:
    
    sage: from sage.features.databases import DatabaseConwayPolynomials
    sage: DatabaseConwayPolynomials().is_present() # optional - database_conway_polynomials
    FeatureTestResult('conway_polynomials', True)


class sage.features.databases.DatabaseCremona(name='cremona', spkg='database_cremona_ellcurve')
    Bases: sage.features.StaticFile

    A Feature which describes the presence of John Cremona's database of elliptic curves.

    INPUT:
    - name -- either 'cremona' (the default) for the full large database or 'cremona_mini' for the small database.

    EXAMPLES:
    
    sage: from sage.features.databases import DatabaseCremona
    sage: DatabaseCremona('cremona_mini').is_present() # optional - database_cremona_ellcurve
    FeatureTestResult('database_cremona_ellcurve', True)

    sage: DatabaseCremona().is_present() # optional - database_cremona_ellcurve
    FeatureTestResult('database_cremona_ellcurve', True)


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: 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('database_knotinfo', True)

sage.features.databases.all_features()
2.11.9 Feature for testing the presence of dvipng

sage.features.dvipng.all_features()

class sage.features.dvipng.dvipng
    Bases: sage.features.Executable
    
    A Feature describing the presence of dvipng

    EXAMPLES:

    sage: from sage.features.dvipng import dvipng
    sage: dvipng().is_present()  # optional - dvipng
    FeatureTestResult('dvipng', True)

2.11.10 Features for testing the presence of fes

class sage.features.fes.LibFES
    Bases: sage.features.join_feature.JoinFeature
    
    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

2.11.11 Feature for testing the presence of ffmpeg

class sage.features.ffmpeg.FFmpeg
    Bases: sage.features.Executable
    
    A Feature describing the presence of ffmpeg

    EXAMPLES:

    sage: from sage.features.ffmpeg import FFmpeg
    sage: FFmpeg().is_present()  # optional - ffmpeg
    FeatureTestResult('ffmpeg', True)

    is_functional()
    
    Return whether command ffmpeg in the path is functional.

    EXAMPLES:
sage: from sage.features.ffmpeg import FFmpeg
sage: FFmpeg().is_functional()  # optional - ffmpeg
FeatureTestResult('ffmpeg', True)

sage.features.ffmpeg.all_features()

## 2.11.12 Features for testing the presence of 4ti2

class sage.features.four_ti_2.FourTi2
    Bases: sage.features.join_feature.JoinFeature

    A Feature describing the presence of the 4ti2 executables.

    EXAMPLES:

    sage: from sage.features.four_ti_2 import FourTi2
    sage: FourTi2().is_present()  # optional - 4ti2
    FeatureTestResult('4ti2', True)

class sage.features.four_ti_2.FourTi2Executable(name)
    Bases: sage.features.Executable

    A Feature for the 4ti2 executables.

sage.features.four_ti_2.all_features()

## 2.11.13 Features for testing the presence of GAP packages

class sage.features.gap.GapPackage(package, **kwds)
    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')

## 2.11.14 Features for testing the presence of various graph generator programs

class sage.features.graph_generators.Benzene
    Bases: sage.features.Executable

    A Feature which checks for the benzene binary.

    EXAMPLES:

    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
gate: Benzene().is_functional()  # optional - benzene
FeatureTestResult('benzene', True)
```

class sage.features.graph_generators.Buckgen
Bases: sage.features.Executable

A Feature which checks for the buckgen binary.

EXAMPLES:

```python
sage: from sage.features.graph_generators import Buckgen
gate: Buckgen().is_present()  # optional - buckgen
FeatureTestResult('buckgen', True)
```

is_functional()
Check whether buckgen works on trivial input.

EXAMPLES:

```python
sage: from sage.features.graph_generators import Buckgen
gate: Buckgen().is_functional()  # optional - buckgen
FeatureTestResult('buckgen', True)
```

class sage.features.graph_generators.Plantri
Bases: sage.features.Executable

A Feature which checks for the plantri binary.

EXAMPLES:

```python
sage: from sage.features.graph_generators import Plantri
gate: Plantri().is_present()  # optional - plantri
gate: Plantri().is_functional()  # optional - plantri
FeatureTestResult('plantri', True)
```

is_functional()
Check whether plantri works on trivial input.

EXAMPLES:

```python
sage: from sage.features.graphGenerators import Plantri
gate: Plantri().is_functional()  # optional - plantri
FeatureTestResult('plantri', True)
```

sage.features.graphGenerators.all_features()
2.11.15 Features for testing the presence of graphviz

```python
class sage.features.graphviz.Graphviz:
    Bases: sage.features.join_feature.JoinFeature

    A Feature describing the presence of the dot, neato, and twopi executables from the graphviz package.

    EXAMPLES:
    sage: from sage.features.graphviz import Graphviz
    sage: Graphviz().is_present()  # optional - graphviz
    FeatureTestResult(\'graphviz\', True)
```

```python
class sage.features.graphviz.dot:
    Bases: sage.features.Executable

    A Feature describing the presence of dot

    EXAMPLES:
    sage: from sage.features.graphviz import dot
    sage: dot().is_present()  # optional - graphviz
    FeatureTestResult(\'dot\', True)
```

```python
class sage.features.graphviz.neato:
    Bases: sage.features.Executable

    A Feature describing the presence of neato

    EXAMPLES:
    sage: from sage.features.graphviz import neato
    sage: neato().is_present()  # optional - graphviz
    FeatureTestResult(\'neato\', True)
```

```python
class sage.features.graphviz.twopi:
    Bases: sage.features.Executable

    A Feature describing the presence of twopi

    EXAMPLES:
    sage: from sage.features.graphviz import twopi
    sage: twopi().is_present()  # optional - graphviz
    FeatureTestResult(\'twopi\', True)
```

2.11.16 Feature for testing the presence of imagemagick

Currently we only check for the presence of convert. When needed other commands like magick, magick-script, convert, mogrify, identify, composite, montage, compare, etc. could be also checked in this module.

```python
class sage.features.imagemagick.Convert:
    Bases: sage.features.Executable

    A Feature describing the presence of convert

    EXAMPLES:
    ```
sage: from sage.features.imagemagick import Convert
sage: Convert().is_present()  # optional - imagemagick
FeatureTestResult('convert', True)

\textbf{is\_functional()}

Return whether command convert in the path is functional.

\textbf{EXAMPLES:}

sage: from sage.features.imagemagick import Convert
sage: Convert().is_functional()  # optional - imagemagick
FeatureTestResult('convert', True)

class \texttt{sage.features.imagemagick.ImageMagick}

\texttt{Bases: sage.features.join\_feature.JoinFeature}

A \texttt{Feature} describing the presence of \texttt{ImageMagick}

Currently, only the availability of \texttt{convert} is checked.

\textbf{EXAMPLES:}

sage: from sage.features.imagemagick import ImageMagick
sage: ImageMagick().is_present()  # optional - imagemagick
FeatureTestResult('imagemagick', True)
sage: ImageMagick().is_functional()  # optional - imagemagick
FeatureTestResult('imagemagick', True)

sage.features.imagemagick.all\_features()

2.11.17 Features for testing whether interpreter interfaces are functional

\textbf{class \texttt{sage.features.interfaces.InterfaceFeature}}(name, module, description)

\texttt{Bases: sage.features.Feature}

A \texttt{Feature} describing whether an \texttt{Interface} is present and functional.

\textbf{class \texttt{sage.features.interfaces.Macaulay2}}(name, module, description)

\texttt{Bases: sage.features.interfaces.InterfaceFeature}

A \texttt{Feature} describing whether \texttt{sage.interfaces.macaulay2.Macaulay2} is present and functional.

\textbf{EXAMPLES:}

sage: from sage.features.interfaces import Macaulay2
sage: Macaulay2().is_present()  # random
FeatureTestResult('macaulay2', False)

\textbf{class \texttt{sage.features.interfaces.Magma}}(name, module, description)

\texttt{Bases: sage.features.interfaces.InterfaceFeature}

A \texttt{Feature} describing whether \texttt{sage.interfaces.magma.Magma} is present and functional.

\textbf{EXAMPLES:}

sage: from sage.features.interfaces import Magma
sage: Magma().is_present()  # random
FeatureTestResult('magma', False)
class sage.features.interfaces.Maple(name, module, description)
Bases: sage.features.interfaces.InterfaceFeature

A Feature describing whether sage.interfaces.maple.Maple is present and functional.

EXAMPLES:

```
sage: from sage.features.interfaces import Maple
sage: Maple().is_present()  # random
FeatureTestResult('maple', False)
```

class sage.features.interfaces.Mathematica(name, module, description)
Bases: sage.features.interfaces.InterfaceFeature

A Feature describing whether sage.interfaces.mathematica.Mathematica is present and functional.

EXAMPLES:

```
sage: from sage.features.interfaces import Mathematica
sage: Mathematica().is_present()  # random
FeatureTestResult('mathematica', False)
```

class sage.features.interfaces.Matlab(name, module, description)
Bases: sage.features.interfaces.InterfaceFeature

A Feature describing whether sage.interfaces.matlab.Matlab is present and functional.

EXAMPLES:

```
sage: from sage.features.interfaces import Matlab
sage: Matlab().is_present()  # random
FeatureTestResult('matlab', False)
```

class sage.features.interfaces.Octave(name, module, description)
Bases: sage.features.interfaces.InterfaceFeature

A Feature describing whether sage.interfaces.octave.Octave is present and functional.

EXAMPLES:

```
sage: from sage.features.interfaces import Octave
sage: Octave().is_present()  # random
FeatureTestResult('octave', False)
```

class sage.features.interfaces.Scilab(name, module, description)
Bases: sage.features.interfaces.InterfaceFeature

A Feature describing whether sage.interfaces.scilab.Scilab is present and functional.

EXAMPLES:

```
sage: from sage.features.interfaces import Scilab
sage: Scilab().is_present()  # random
FeatureTestResult('scilab', False)
```

sage.features.interfaces.all_features()
Return features corresponding to interpreter interfaces.

EXAMPLES:
sage: from sage.features.interfaces import all_features
sage: list(all_features())

[Feature('magma'),
 Feature('matlab'),
 Feature('mathematica'),
 Feature('maple'),
 Feature('macaulay2'),
 Feature('octave'),
 Feature('scilab')]

2.11.18 Feature for testing if the Internet is available

class sage.features.internet.Internet
    Bases: sage.features.Feature

    A Feature describing if Internet is available.

    Failure of connecting to the site “https://www.sagemath.org” within a second is regarded as internet being not available.

    EXAMPLES:

    sage: from sage.features.internet import Internet
    sage: Internet()
    Feature('internet')

sage.features.internet.all_features()

2.11.19 Feature for testing the presence of kenzo

class sage.features.kenzo.Kenzo
    Bases: sage.features.Feature

    A Feature describing the presence of Kenzo.

    EXAMPLES:

    sage: from sage.features.kenzo import Kenzo
    sage: Kenzo().is_present()  # optional - kenzo
    FeatureTestResult('kenzo', True)

sage.features.kenzo.all_features()

2.11.20 Features for testing the presence of latex and equivalent programs

class sage.features.latex.LaTeXPackage\(\text{name, filename, **kwds}\)
    Bases: sage.features.latex.TeXFile

    A sage.features.Feature describing the presence of a LaTeX package (.sty file).

    EXAMPLES:
```python
sage: from sage.features.latex import LaTeXPackage
sage: LaTeXPackage('graphics').is_present()  # optional - pdflatex
FeatureTestResult('latex_package_graphics', True)
```

```python
class sage.features.latex.TexFile(name, filename, **kwds)
    Bases: sage.features.StaticFile

    A sage.features.Feature describing the presence of a TeX file

    EXAMPLES:
```  
```python
sage: from sage.features.latex import TexFile
sage: TexFile('x', 'x.tex').is_present()  # optional: pdflatex
FeatureTestResult('x', True)
sage: TexFile('nonexisting', 'xxxxxx-nonexisting-file.tex').is_present()  # optional - pdflatex
FeatureTestResult('nonexisting', False)
```

```python
absolute_path()
    The absolute path of the file.

    EXAMPLES:
```  
```python
sage: from sage.features.latex import TexFile
sage: feature = TexFile('latex_class_article', 'article.cls')
sage: feature.absolute_path()  # optional - pdflatex
'.../latex/base/article.cls'
```

```python
sage.features.latex.all_features()
```

```python
class sage.features.latex.latex
    Bases: sage.features.Executable

    A Feature describing the presence of latex

    EXAMPLES:
```  
```python
sage: from sage.features.latex import latex
sage: latex().is_present()  # optional - latex
FeatureTestResult('latex', True)
```

```python
is_functional()
    Return whether latex in the path is functional.

    EXAMPLES:
```  
```python
sage: from sage.features.latex import latex
sage: latex().is_functional()  # optional - latex
FeatureTestResult('latex', True)
```

```python
class sage.features.latex.lualatex
    Bases: sage.features.Executable

    A Feature describing the presence of lualatex

    EXAMPLES:
```
sage: from sage.features.latex import lualatex
sage: lualatex().is_present()  # optional - lualatex
FeatureTestResult('lualatex', True)

class sage.features.latex.pdflatex
Bases: sage.features.Executable

A Feature describing the presence of pdflatex

EXAMPLES:

sage: from sage.features.latex import pdflatex
sage: pdflatex().is_present()  # optional - pdflatex
FeatureTestResult('pdflatex', True)

class sage.features.latex.xelatex
Bases: sage.features.Executable

A Feature describing the presence of xelatex

EXAMPLES:

sage: from sage.features.latex import xelatex
sage: xelatex().is_present()  # optional - xelatex
FeatureTestResult('xelatex', True)

2.11.21 Features for testing the presence of latte_int

class sage.features.latte.Latte
Bases: sage.features.join_feature.JoinFeature

A Feature describing the presence of the LattE binaries which comes as a part of latte_int.

EXAMPLES:

sage: from sage.features.latte import Latte
sage: Latte().is_present()  # optional - latte_int
FeatureTestResult('latte_int', True)

class sage.features.latte.Latte_count
Bases: sage.features.Executable

Feature for the executable count from the LattE suite.

class sage.features.latte.Latte_integrate
Bases: sage.features.Executable

Feature for the executable integrate from the LattE suite.

sage.features.latte.all_features()
2.11.22 Feature for testing the presence of lrslib

class sage.features.lrs.Lrs
   Bases: sage.features.Executable

   A Feature describing the presence of the lrs binary which comes as a part of lrslib.

   EXAMPLES:

   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:

   sage: from sage.features.lrs import Lrs
   sage: Lrs().is_functional()  # optional - lrslib
   FeatureTestResult('lrslib', True)

sage.features.lrs.all_features()

2.11.23 Features for testing the presence of mcqd

class sage.features.mcqd.Mcq
   Bases: sage.features.join_feature.JoinFeature

   A Feature describing the presence of mcqd

   EXAMPLES:

   sage: from sage.features.mcqd import Mcq
   sage: Mcq().is_present()  # optional - mcqd
   FeatureTestResult('mcqd', True)

sage.features.mcqd.all_features()

2.11.24 Feature for testing the presence of meataxe

class sage.features.meataxe.Meataxe
   Bases: sage.features.join_feature.JoinFeature

   A Feature describing the presence of meataxe.

   EXAMPLES:

   sage: from sage.features.meataxe import Meataxe
   sage: Meataxe().is_present()  # optional - meataxe
   FeatureTestResult('meataxe', True)

sage.features.meataxe.all_features()
2.11.25 Features for testing the presence of MixedIntegerLinearProgram backends

class sage.features.mip_backends.COIN
    Bases: sage.features.join_feature.JoinFeature
    A Feature describing whether the MixedIntegerLinearProgram backend COIN is available.

class sage.features.mip_backends.CPLEX
    Bases: sage.features.mip_backends.MIPBackend
    A Feature describing whether the MixedIntegerLinearProgram backend CPLEX is available.

class sage.features.mip_backends.Gurobi
    Bases: sage.features.mip_backends.MIPBackend
    A Feature describing whether the MixedIntegerLinearProgram backend Gurobi is available.

class sage.features.mip_backends.MIPBackend(name, spkg=None, url=None, description=None)
    Bases: sage.features.Feature
    A Feature describing whether a MixedIntegerLinearProgram backend is available.
sage.features.mip_backends.all_features()

2.11.26 Feature for testing the presence of pynormaliz

class sage.features.normaliz.PyNormaliz
    Bases: sage.features.join_feature.JoinFeature
    A Feature describing the presence of the Python package PyNormaliz.
    EXAMPLES:
    sage: from sage.features.normaliz import PyNormaliz
    sage: PyNormaliz().is_present()    # optional - pynormaliz
    FeatureTestResult('pynormaliz', True)
sage.features.normaliz.all_features()

2.11.27 Feature for testing the presence of pandoc

class sage.features.pandoc.Pandoc
    Bases: sage.features.Executable
    A Feature describing the presence of pandoc
    EXAMPLES:
    sage: from sage.features.pandoc import Pandoc
    sage: Pandoc().is_present()    # optional - pandoc
    FeatureTestResult('pandoc', True)
sage.features.pandoc.all_features()
2.11.28 Feature for testing the presence of pdf2svg

sage.features.pdf2svg.all_features()

class sage.features.pdf2svg.pdf2svg
    Bases: sage.features.Executable
    
    A Feature describing the presence of pdf2svg

    EXAMPLES:

    sage: from sage.features.pdf2svg import pdf2svg
    sage: pdf2svg().is_present()  # optional - pdf2svg
    FeatureTestResult('pdf2svg', True)

2.11.29 Feature for testing the presence of the Python interface to polymake

class sage.features.polymake.JuPyMake
    Bases: sage.features.join_feature.JoinFeature
    
    A Feature describing the presence of the JuPyMake module, a Python interface to the polymake library.

    EXAMPLES:

    sage: from sage.features.polymake import JuPyMake
    sage: JuPyMake().is_present()  # optional - jupymake
    FeatureTestResult('jupymake', True)

sage.features.polymake.all_features()

2.11.30 Features for testing the presence of rubiks

class sage.features.rubiks.Rubiks
    Bases: sage.features.join_feature.JoinFeature
    
    A Feature describing the presence of cu2, cubex, dikcube, mcube, optimal, and size222.

    EXAMPLES:

    sage: from sage.features.rubiks import Rubiks
    sage: Rubiks().is_present()  # optional - rubiks
    FeatureTestResult('rubiks', True)

sage.features.rubiks.all_features()

class sage.features.rubiks.cu2
    Bases: sage.features.Executable
    
    A Feature describing the presence of cu2

    EXAMPLES:

    sage: from sage.features.rubiks import cu2
    sage: cu2().is_present()  # optional - rubiks
    FeatureTestResult('cu2', True)
class sage.features.rubiks.cubex
    Bases: sage.features.Executable
    A Feature describing the presence of cubex

    EXAMPLES:

    sage: from sage.features.rubiks import cubex
    sage: cubex().is_present()  # optional - rubiks
    FeatureTestResult('cubex', True)

class sage.features.rubiks.dikcube
    Bases: sage.features.Executable
    A Feature describing the presence of dikcube

    EXAMPLES:

    sage: from sage.features.rubiks import dikcube
    sage: dikcube().is_present()  # optional - rubiks
    FeatureTestResult('dikcube', True)

class sage.features.rubiks.mcube
    Bases: sage.features.Executable
    A Feature describing the presence of mcube

    EXAMPLES:

    sage: from sage.features.rubiks import mcube
    sage: mcube().is_present()  # optional - rubiks
    FeatureTestResult('mcube', True)

class sage.features.rubiks.optimal
    Bases: sage.features.Executable
    A Feature describing the presence of optimal

    EXAMPLES:

    sage: from sage.features.rubiks import optimal
    sage: optimal().is_present()  # optional - rubiks
    FeatureTestResult('optimal', True)

class sage.features.rubiks.size222
    Bases: sage.features.Executable
    A Feature describing the presence of size222

    EXAMPLES:

    sage: from sage.features.rubiks import size222
    sage: size222().is_present()  # optional - rubiks
    FeatureTestResult('size222', True)
2.11.31 Features for testing the presence of tdlib

class sage.features.tdlib.Tdlib
    Bases: sage.features.join_feature.JoinFeature

    A Feature describing the presence of the tdlib.

sage.features.tdlib.all_features()
3.1 Cython support functions

AUTHORS:

- William Stein (2006-01-18): initial version

sage.misc.cython.compile_and_load(code, **kwds)

INPUT:

- code – string containing code that could be in a .pyx file that is attached or put in a %cython block in the notebook.

OUTPUT: a module, which results from compiling the given code and importing it

EXAMPLES:

```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](https://trac.sagemath.org/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*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')
sage: 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')
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.cythonsanitize(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

```python
class sage.misc.inline_fortran.InlineFortran(globals=None)
    Bases: object

    add_library(s)
    add_library_path(s)
    eval(x, globals=None, locals=None)

Compile fortran code x and adds the functions in it to globals.

INPUT:
- x – Fortran code
- globals – a dict to which to add the functions from the fortran module
- locals – ignored

EXAMPLES:
sage: code = '''
....: C FILE: FIB1.F
....: SUBROUTINE FIB(A,N)
....:  C
....:  C   CALCULATE FIRST N FIBONACCI NUMBERS
....:  C
....:     INTEGER N
....:     REAL*8 A(N)
....:     DO I=1,N
....:         IF (I.EQ.1) THEN
....:             A(I) = 0.0D0
....:         ELSEIF (I.EQ.2) THEN
....:             A(I) = 1.0D0
....:         ELSE
....:             A(I) = A(I-1) + A(I-2)
....:         ENDIF
....:     ENDDO
....: END
....: C END FILE FIB1.F
....: '''
sage: fortran(code, globals())
sage: import numpy
da = numpy.array(range(10), dtype=float)
sage: fib(a, 10)
da
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)
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)
```
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(tokens)}

Returns an expr, or a (name, expr) tuple corresponding to a single function call argument.

**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(tokens)}

Returns a list, dict pair.

**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}
\begin{verbatim}
 sage: p.p_args(Tokenizer("1, 2, a = 1+5^2"))
 ([(1, 2), {'a': 26}])
\end{verbatim}

\textbf{p_atom}(\textit{tokens})

Parse an atom. This is either a parenthesized expression, a function call, or a literal name/int/float.

\textbf{EXAMPLES:}
\begin{verbatim}
 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()"))
 ((), {})
\end{verbatim}

\textbf{p_eqn}(\textit{tokens})

Parse an equation or expression.

This is the top-level node called by the \texttt{parse} function.

\textbf{EXAMPLES:}
\begin{verbatim}
 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
\end{verbatim}

\textbf{p_expr}(\textit{tokens})

Parse a list of one or more terms.

\section{A parser for symbolic equations and expressions}

257
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"))
-d^2 + a - b + 4*c
sage: p.p_expr(Tokenizer("a - -3"))
a + 3
sage: p.p_expr(Tokenizer("a + 1 == b"))
a + 1
```

**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

```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:
from sage.misc.parser import Parser, Tokenizer

R.<t> = ZZ[['t']]
p = Parser(make_var={'t': t})
p.p_factor(Tokenizer("-(1+t)^-1"))
-1 + t - t^2 + t^3 - t^4 + t^5 - t^6 + t^7 - t^8 + t^9 - t^10 + t^11 - t^12 + t^13 - t^14 + t^15 - t^16 + t^17 - t^18 + t^19 + O(t^20)
p.p_factor(Tokenizer("t**2"))
t^2
p.p_power(Tokenizer("2^3^2")) == 2^9
True

p_sequence(tokens)
Parse a (possibly nested) set of lists and tuples.

EXAMPLES:

from sage.misc.parser import Parser, Tokenizer
p = Parser(make_var=var)
p.p_sequence(Tokenizer("[1+2,0]"))
[[3, 0]]
p.p_sequence(Tokenizer("(1,2,3) , [1+a, 2+b, (3+c), (4+d,)]"))
[(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:

from sage.misc.parser import Parser, Tokenizer
p = Parser(make_var=var)
p.p_term(Tokenizer("a*b"))
a*b
p.p_term(Tokenizer("a * b / c * d"))
a*b/d/c
p.p_term(Tokenizer("-a * b + c"))
-a*b
p.p_term(Tokenizer("a*(b-c)^2"))
a*(b - c)^2
p.p_term(Tokenizer("-3a"))
-3*a

p_tuple(tokens)
Parse a tuple of items.

EXAMPLES:
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:

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:

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:

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:

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:

sage: Tokenizer("+-*/^(),=</>\[\]{}").test()

Two-character comparisons accepted are:

```python
tokenizer = Tokenizer("<= >= != == **")
tokenizer.test()
```

Integers are strings of 0-9:

```python
tokenizer = Tokenizer("1 123 9879834759873452908375013")
tokenizer.test()
```

Floating point numbers can contain a single decimal point and possibly exponential notation:

```python
tokenizer = Tokenizer("1. .01 1e3 1.e-3")
tokenizer.test()
```

Note that negative signs are not attached to the token:

```python
tokenizer = Tokenizer("-1 -1.2")
tokenizer.test()
```

Names are alphanumeric sequences not starting with a digit:

```python
tokenizer = Tokenizer("a a1 _a_24")
tokenizer.test()
```

There is special handling for matrices:

```python
tokenizer = Tokenizer("matrix(a)")
tokenizer.test()
```

Anything else is an error:

```python
tokenizer = Tokenizer("&@~")
tokenizer.test()
```

No attempt for correctness is made at this stage:

```python
tokenizer = Tokenizer("( ) ( 5e5")
tokenizer.test()
```

```python
tokenizer = Tokenizer("?$%")
tokenizer.test()
```

`backtrack()` function:

```
backtrack()
```

Put self in such a state that the subsequent call to `next()` will return the same as if `next()` had not been called.

Currently, one can only backtrack once.

**EXAMPLES:**

```python
from sage.misc.parser import Tokenizer, token_to_str
t = Tokenizer("a+b")
token_to_str(t.next())
'NAME'
token_to_str(t.next())
'+'
```

(continues on next page)
last()  
Returns the last token seen.

EXAMPLES:

```python
sage: from sage.misc.parser import Tokenizer, token_to_str
sage: t = Tokenizer("3a")
```

```python
sage: token_to_str(t.next())  
'INT'
```

```python
sage: token_to_str(t.last())  
'INT'
```

```python
sage: token_to_str(t.next())  
'NAME'
```

```python
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")
```

```python
sage: token_to_str(t.next())  
'NAME'
```

```python
sage: t.last_token_string()  
'a'
```

```python
sage: token_to_str(t.next())  
'-'
```

```python
sage: token_to_str(t.next())  
'FLOAT'
```

```python
sage: t.last_token_string()  
'1e5'
```

next()  
Returns the next token in the string.

EXAMPLES:

```python
sage: from sage.misc.parser import Tokenizer, token_to_str
sage: t = Tokenizer("a+3")
```

```python
sage: token_to_str(t.next())  
'NAME'
```

```python
sage: token_to_str(t.next())  
'+'
```

```python
sage: token_to_str(t.next())  
'INT'
```

```python
sage: token_to_str(t.next())  
'EOS'
```
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'
```

```python
sage: token_to_str(t.next())
'NAME'
```

```python
sage: token_to_str(t.peek())
'+'
```

```python
sage: token_to_str(t.peek())
'+'
```

```python
sage: token_to_str(t.next())
'+'
```

reset(pos=0)

Reset the tokenizer to a given position.

EXAMPLES:

```python
sage: from sage.misc.parser import Tokenizer
sage: t = Tokenizer("a+b*c")
```

```python
sage: t.test()
['NAME(a)', '+', 'NAME(b)', '*', 'NAME(c)']
```

```python
sage: t.test()
[]
```

```python
sage: t.reset()
```

```python
sage: t.test()
['NAME(a)', '+', 'NAME(b)', '*', 'NAME(c)']
```

```python
sage: t.reset(3)
```

```python
sage: t.test()
['*', 'NAME(c)']
```

No care is taken to make sure we don’t jump in the middle of a token:

```python
sage: t = Tokenizer("12345+a")
```

```python
sage: t.test()
['INT(12345)', '+', 'NAME(a)']
```

```python
sage: t.reset(2)
```

```python
sage: t.test()
['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")
```

```python
sage: t.test()
['NAME(a)', 'NAME(b)', 'INT(3)']
```

(continues on next page)
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")
```

3.4 Evaluating Python code without any preparsing

```python
class sage.misc.python.Python
    Bases: object

    Allows for evaluating a chunk of code without any preparsing.

eval(x, globals, locals=None)
    Evaluate x with given globals; locals is completely ignored.
    This is specifically meant for evaluating code blocks with %python in the notebook.

    INPUT:
    • x – a string
    • globals – a dictionary
    • locals – completely IGNORED

    EXAMPLES:
```
```python
sage: from sage.misc.python import Python
sage: python = Python()
```

Any variables that are set during evaluation of the block will propagate to the globals dictionary.
sage: python.eval('a=5\nb=7\na+b', globals())
12
'

sage: b
7

The `locals` variable is ignored – it is there only for completeness. It is ignored since otherwise the following will not work:

```
sage: python.eval("def foo():\n    return 'foo'\nprint(foo())\n    \n    \ndef mumble():\n        print('mumble {}\n    format(foo()))\n    \n    mumble()", globals())
foo
mumble foo
'
```

### 3.5 Evaluating a String in Sage

`sage.misc.sage_eval.sage_eval(source, locals=None, cmds='', preparse=True)`

Obtain a Sage object from the input string by evaluating it using Sage. This means calling `eval` after preparsing and with globals equal to everything included in the scope of `from sage.all import *`).

**INPUT:**

- `source` - a string or object with a `_sage_` method
- `locals` - evaluate in namespace of `sage.all` plus the `locals` dictionary
- `cmds` - string: sequence of commands to be run before source is evaluated.
- `preparse` - (default: True) if True, parse the string expression.

**EXAMPLES:** This example illustrates that preparsing is applied.

```
sage: eval('2^3')
1
sage: sage_eval('2^3')
8
```

However, preparsing can be turned off.

```
sage: sage_eval('2^3', preparse=False)
1
```

Note that you can explicitly define variables and pass them as the second option:

```
sage: x = PolynomialRing(RationalField(),"x").gen()
sage: sage_eval('x^2+1', locals={'x':x})
x^2 + 1
```

This example illustrates that evaluation occurs in the context of `from sage.all import *`. Even though `bernoulli` has been redefined in the local scope, when calling `sage_eval` the default value meaning of `bernoulli` is used. Likewise for QQ below.
sage: bernoulli = lambda x : x^2
sage: bernoulli(6)
36
sage: eval('bernoulli(6)')
36
sage: sage_eval('bernoulli(6)')
1/42

sage: QQ = lambda x : x^2
sage: QQ(2)
4
sage: sage_eval('QQ(2)')
2
sage: parent(sage_eval('QQ(2)'))
Rational Field

This example illustrates setting a variable for use in evaluation.

sage: x = 5
sage: eval('4/3 + x', {'x': 25})
26
sage: sage_eval('4/3 + x', locals={'x': 25})
79/3

You can also specify a sequence of commands to be run before the expression is evaluated:

sage: sage_eval('p', cmds='K.<x> = QQ[]
    p = x^2 + 1')
x^2 + 1

If you give commands to execute and a dictionary of variables, then the dictionary will be modified by assignments in the commands:

sage: vars = {}
sage: sage_eval('None', cmds='y = 3', locals=vars)
sage: vars['y'], parent(vars['y'])
(3, Integer Ring)

You can also specify the object to evaluate as a tuple. A 2-tuple is assumed to be a pair of a command sequence and an expression; a 3-tuple is assumed to be a triple of a command sequence, an expression, and a dictionary holding local variables. (In this case, the given dictionary will not be modified by assignments in the commands.)

sage: sage_eval((f(x) = x^2, 'f(3)'))
9
sage: vars = {'rt2': sqrt(2.0)}
sage: sage_eval(('rt2 += 1', 'rt2', vars))
2.41421356237309
sage: vars['rt2']
1.41421356237310

This example illustrates how sage_eval can be useful when evaluating the output of other computer algebra systems.
sage: R.<x> = PolynomialRing(RationalField())
sage: gap.eval('R:=PolynomialRing(Rationals,["x"]);; f:=x^2+1;')
'Rationals[x]
ff = gap.eval('x:=IndeterminatesOfPolynomialRing(R);; f:=x^2+1;')
sage: sage_eval(ff, locals={'x':x})
x^2 + 1
eval(ff)
Traceback (most recent call last):
  ...RuntimeError: Use ** for exponentiation, not ^, which means xor in Python, and has the wrong precedence.

Here you can see eval simply will not work but sage_eval will.

sage.misc.sage_eval.sageobj(x, vars=None)
Return a native Sage object associated to x, if possible and implemented.

If the object has an _sage_ method it is called and the value is returned. Otherwise str is called on the object, and all preparsing is applied and the resulting expression is evaluated in the context of from sage.all import *.

To evaluate the expression with certain variables set, use the vars argument, which should be a dictionary.

EXAMPLES:

sage: type(sageobj(gp('34/56')))
<class '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)
<class 'sage.rings.integer.Integer'>

This illustrates interfaces:

sage: f = gp('2/3')
sage: type(f)
<class 'sage.interfaces.gp.GpElement'>
sage: f._sage_
2/3
sage: type(f._sage_())
<class 'sage.rings.rational.Rational'>
sage: a = gap(939393/2433)
sage: a._sage_
313131/811
sage: type(a._sage_())
<class 'sage.rings.rational.Rational'>

3.5. Evaluating a String in Sage
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"
if [ $? -eq 0 ]; then
echo "good"
fi''')
```
4.1 Formatted Output

4.1.1 Symbols for Character Art

This module defines single- and multi-line symbols.

EXAMPLES:

```python
sage: from sage.typeset.symbols import *
sage: symbols = ascii_art('')
sage: for i in range(1, 5):
    ....:    symbols += ascii_left_parenthesis.character_art(i)
    ....:    symbols += ascii_art(' ')
    ....:    symbols += ascii_right_parenthesis.character_art(i)
    ....:    symbols += ascii_art(' ')
sage: for i in range(1, 5):
    ....:    symbols += ascii_left_square_bracket.character_art(i)
    ....:    symbols += ascii_art(' ')
    ....:    symbols += ascii_right_square_bracket.character_art(i)
    ....:    symbols += ascii_art(' ')
sage: for i in range(1, 5):
    ....:    symbols += ascii_left_curly_brace.character_art(i)
    ....:    symbols += ascii_art(' ')
    ....:    symbols += ascii_right_curly_brace.character_art(i)
    ....:    symbols += ascii_art(' ')
sage: symbols
( ) [ ] { }
( ) ( ) [ ] [ ] { } { }
( ) ( ) ( ) [ ] [ ] [ ] { } { } { }
( ) ( ) ( ) ( ) [ ] [ ] [ ] [ ] { } { } { } { }
sage: symbols = unicode_art('')
sage: for i in range(1, 5):
    ....:    symbols += unicode_left_parenthesis.character_art(i)
    ....:    symbols += unicode_art(' ')
    ....:    symbols += unicode_right_parenthesis.character_art(i)
    ....:    symbols += unicode_art(' ')
sage: for i in range(1, 5):
    ....:    symbols += unicode_left_square_bracket.character_art(i)
```

(continues on next page)
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

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(character, top, extension, bottom, middle=None, middle_top=None, middle_bottom=None, top_2=None, bottom_2=None)

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
    @classmethod empty()
    Return the empty character art object

    EXAMPLES:
    
    sage: from sage.typeset.ascii_art import AsciiArt
    sage: AsciiArt.empty()

    get_baseline()
    Return the line where the baseline is, for example:

    5 4
    14*x + 5*x

    the baseline has at line 0 and

    { o  }
    { \ : 4 }
    { o  }

    has at line 1.

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

    This method is deprecated, as its output is an implementation detail. The mere breakpoints of a character
    art element do not reflect the best way to split it if nested structures are involved. For details, see trac ticket
    #29204.

    For example 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()
```

```
[6]
```

See https://trac.sagemath.org/29204 for details.

```
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

```python
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:
    sage: from sage.typeset.ascii_art import _ascii_art_factory as factory
    sage: type(factory)
    <class 'sage.typeset.character_art_factory.CharacterArtFactory'>

    build(obj, baseline=None)
    Construct a character art representation.
    INPUT:
    • obj – anything; the object whose ascii art representation we want
    • baseline – (optional) the baseline of the object
    OUTPUT:
    Character art object.
    EXAMPLES:
    sage: result = ascii_art(integral(exp(x+x^2)/(x+1), x))
    ...
    sage: result
    /  
    |   2
    | x + x
    | e
    | ------ dx
    | x + 1
    | 

    build_container(content, left_border, right_border, baseline=0)
    Return character art for a container.
```
INPUT:

- **content** – *CharacterArt*; the content of the container, usually comma-separated entries
- **left_border** – *CompoundSymbol*; the left border of the container
- **right_border** – *CompoundSymbol*; the right border of the container
- **baseline** – (default: 0) the baseline of the object

**build_dict**(*d*, **baseline**=0)

Return a character art output of a dictionary.

**build_empty**()

Return the empty character art object

**OUTPUT:**

Character art instance.

**EXAMPLES:**

```sage
from sage.typeset.ascii_art import _ascii_art_factory as factory
sage: str(factory.build_empty())
```

**build_from_magic_method**(*obj*, **baseline**=None)

Return the character art object created by the object’s magic method

**OUTPUT:**

Character art instance.

**EXAMPLES:**

```sage
from sage.typeset.ascii_art import _ascii_art_factory as factory
go: out = factory.build_from_magic_method(identity_matrix(2)); out
[1 0]
[0 1]
sage: type(out)
<class 'sage.typeset.ascii_art.AsciiArt'>
```

**build_from_string**(*obj*, **baseline**=0)

Return the character art object created from splitting the object’s string representation.

**INPUT:**

- **obj** – utf-8 encoded byte string or unicode
- **baseline** – (default: 0) the baseline of the object

**OUTPUT:**

Character art instance.

**EXAMPLES:**

```sage
from sage.typeset.ascii_art import _ascii_art_factory as factory
go: out = factory.build_from_string('a
bb
ccc'
 sage: out + out + out
a
a
a
bb
bb
bb
cccccccccccc
```

(continues on next page)
sage: type(out)
<class 'sage.typeset.ascii_art.AsciiArt'>

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))
\[
\begin{bmatrix}
1 & 1 \\
[1 & 0] & [1 & 0] & [1 & 0] \\
[0 & 1] & [0 & 1] & [0 & 1]
\end{bmatrix}
\]

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))
2
n *log(x)
---------
pi
sage: ascii_art(list(Partitions(6)))
[ *,]
[ ** * ,]
[ *** ** * *
[ **** **** * *** * * * *
[ ****** ***** * ******* * * * * *
[ ******* *, ****** * **** ** * *** * *
```

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

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

(continues on next page)
```python
sage: shell.run_cell('%display simple')
sage: shell.quit()
```
class sage.typeset.ascii_art.AsciiArt(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
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:**

```sage
result = ascii_art(integral(exp(x+x^2)/(x+1), x))
```

```sage
result
```

```
/ 2 | x + x | e
| -------- dx |
| x + 1 |
```

We can specify a separator object:

```sage
ident = lambda n: identity_matrix(ZZ, n)
sage: ascii_art(ident(1), ident(2), ident(3), sep=' : ')
```

```
[1 0 0]
[1 0]  [0 1 0]
[1] : [0 1] : [0 0 1]
```

We can specify the baseline:

```sage: ascii_art(ident(2), baseline=-1) + ascii_art(ident(3))
```

```
[1 0] [1 0 0] [0 1 0]
[0 1]  [0 1 0] [0 0 1]
```

We can determine the baseline of the separator:

```sage: ascii_art(ident(1), ident(2), ident(3), sep=' -- ', sep_baseline=-1)
```

```
[1 0 0]
-- [1 0] -- [0 1 0]
[1]  [0 1]  [0 0 1]
```

If specified, the *sep_baseline* overrides the baseline of an ascii art separator:

```sage: sep_line = ascii_art('
'.join(' ' * for _ in range(6)), baseline=6)
sage: ascii_art(*Partitions(6), separator=sep_line, sep_baseline=0)
```

```
* | * | * | * | * | * | * | *
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
```

280 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)
    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:

    sage: i = var('i')
    sage: unicode_art(sum(pi^i/factorial(i)*x^i, i, 0, oo))
    𝜋 · 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:

    sage: result = unicode_art(integral(exp(sqrt(x))/(x+pi), x))
    ...  
    sage: result
    ∫ x
    e
    dx
    x + π

    sage: ident = lambda n: identity_matrix(ZZ, n)
    sage: unicode_art(ident(1), ident(2), ident(3), sep=' : ')
    (1) :
    | 1 0 |
    | 0 1 |
    | 0 0 |
If specified, the `sep_baseline` overrides the baseline of an unicode art separator:

```python
sage: sep_line = unicode_art('|'.join(' ' for _ in range(5)), baseline=5)
sage: unicode_art(*AlternatingSignMatrices(3),
          separator=sep_line, sep_baseline=1)
```

```
\[
\begin{array}{ccc}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{array}
\quad
\begin{array}{ccc}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{array}
\quad
\begin{array}{ccc}
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 1 & 1
\end{array}
\quad
\begin{array}{ccc}
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 1
\end{array}
\quad
\begin{array}{ccc}
0 & 1 & 0 \\
0 & 0 & 1 \\
0 & 0 & 1
\end{array}
\quad
\begin{array}{ccc}
0 & 1 & 0 \\
0 & 0 & 1 \\
0 & 0 & 1
\end{array}
\end{array}
```

```
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'
```

```python
sage: from sage.typeset.unicode_characters import unicode_bigotimes
sage: unicode_bigotimes
'⨂'
```

(continues on next page)
4.1.7 Repr formatting support

\begin{verbatim}
  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
\end{verbatim}
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: a space 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', t^2 + 2)])
'-t*a + (t-2)*s + (t^2+2)'
```

Examples for scalar_mult:
Examples for scalar_mult and is_latex:

```python
sage: repr_lincomb([('a',-1), ('b',2), ('c',3)], is_latex=True, scalar_mult='**')
'-a + 2*b - 3*c'
sage: repr_lincomb([('a',-1), ('b',-1), ('c',-3)], is_latex=True, latex_scalar_mult='\rightarrow')
'-2*a + b + 3*c'
sage: repr_lincomb([('a',-2), ('b',-1), ('c',-3)], is_latex=True, latex_scalar_mult='\rightarrow')
'-2*a - b - 3*c'
```

Examples for strip_one:

```python
sage: repr_lincomb([('a',1), (1,-2), ('3',3)], strip_one=True)
'a - 1 + 3*3'
sage: repr_lincomb([('a',-1), (1,1), ('3',3)], strip_one=True)
'a + 1 + 3*3'
sage: repr_lincomb([('a',1), (1,-2), ('3',3)], strip_one=True)
'a - 1 + 3*3'
sage: repr_lincomb([('a',1), (1,-1), ('3',3)], strip_one=True)
'a - 1 + 3*3'
```

Examples for repr_monomial:

```python
sage: repr_lincomb([('a',i), ('b',2), ('c',3)], repr_monomial=lambda s: s+"1")
'a1 + 2*b1 + 3*c1'
```

### 4.1.8 Sage Input Formatting

This module provides the function `sage_input()` that takes an arbitrary sage value and produces a sequence of commands that, if typed at the `sage:` prompt, will recreate the value. If this is not implemented for a particular value, then an exception is raised instead. This might be useful in understanding a part of Sage, or for debugging. For instance, if you have a value produced in a complicated way in the middle of a debugging session, you could use `sage_input()` to find a simple way to produce the same value. We attempt to produce commands that are readable and idiomatic.:
R.<x> = RR[]
x^2 + 6.2831853071795862*x + 9.869604401089358

With `verify=True`, `sage_input()` also verifies the results, by calling `sage_eval()` on the result and verifying that it is equal to the input:

```python
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:

```python
sage: sage_input(5, verify=True)
# Verified
5
sage: sage_input(5, preparse=False)
ZZ(5)
sage: sage_input(5, preparse=None)
ZZ(5)
sage: sage_input(5r, verify=True)
# Verified
5r
sage: sage_input(5r, preparse=False)
5
sage: sage_input(5r, preparse=None)
int(5)
```

Adding `sage_input()` support to your own classes is straightforward. You need to add a `_sage_input_()` method which returns a `SageInputExpression` (henceforth abbreviated as SIE) which will reconstruct this instance of your class.

A `_sage_input_` method takes two parameters, conventionally named `sib` and `coerced`. The first argument is a `SageInputBuilder`; it has methods to build SIEs. The second argument, `coerced`, is a boolean. This is only useful if your class is a subclass of `Element` (although it is always present). If `coerced` is `False`, then your method must generate an expression which will evaluate to a value of the correct type with the correct parent. If `coerced` is `True`, then your method may generate an expression of a type that has a canonical coercion to your type; and if `coerced` is `2`, then your method may generate an expression of a type that has a conversion to your type.

Let's work through some examples. We'll build a sequence of functions that would be acceptable as `_sage_input_` methods for the `Rational` class.

Here's the first and simplest version:

```python
sage: def qq_sage_input_v1(self, sib, coerced):
....:     return sib(self.numerator())/sib(self.denominator())
```

We see that given a `SageInputBuilder` `sib`, you can construct a SIE for a value `v` simply with `sib(v)`, and you can construct a SIE for a quotient with the division operator. Of course, the other operators also work, and so do function calls, method calls, callbacks, etc.

We'll test with the following code, which you don't need to understand. (It produces a list of 8 results, showing the formatted versions of `-5/7` and `3`, with the preparser either enabled or disabled and either with or without an automatic coercion to `QQ`):
sage: from sage.misc.sage_input import SageInputBuilder
sage: def test_qq_formatter(fmt):
....:     results = []
....:     for v in [-5/7, QQ(3)]:
....:         for pp in [False, True]:
....:             for coerced in [False, True]:
....:                 sib = SageInputBuilder(preparse=pp)
....:                 results.append(sib.result(fmt(v, sib, coerced)))
....:     return results

sage: test_qq_formatter(qq_sage_input_v1)
[-ZZ(5)/ZZ(7), -ZZ(5)/ZZ(7), -5/7, -5/7, ZZ(3)/ZZ(1), ZZ(3)/ZZ(1), 3/1, 3/1]

Let's try for some shorter, perhaps nicer-looking output. We'll start by getting rid of the ZZ in the denominators; even without the preparser, -ZZ(5)/7 == -ZZ(5)/ZZ(7).

sage: def qq_sage_input_v2(self, sib, coerced):
....:     return sib(self.numerator())/sib.int(self.denominator())

The \texttt{int} method on \texttt{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 \texttt{Integer}, an \texttt{int}, or a \texttt{long}.)

sage: test_qq_formatter(qq_sage_input_v2)
[-ZZ(5)/7, -ZZ(5)/7, -5/7, -5/7, ZZ(3)/1, ZZ(3)/1, 3/1, 3/1]

Next let us get rid of the divisions by 1. These are more complicated, since if we are not careful we will get results in \texttt{Z} instead of \texttt{Q}:

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()))
....:     else:
....:         return sib(self.numerator())/sib.int(self.denominator())

We see that the method\texttt{name} method gives an SIE representing a sage constant or function.:

sage: test_qq_formatter(qq_sage_input_v3)
[-ZZ(5)/7, -ZZ(5)/7, -5/7, -5/7, QQ(3), 3, QQ(3), 3]

This is the prettiest output we're going to get, but let's make one further refinement. Other \texttt{._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 \texttt{sage_input(polygen(QQ) - 2/3)} would produce \texttt{x} + (-2/3); if we change to the following code, then we would get \texttt{x} - 2/3 instead.:

sage: def qq_sage_input_v4(self, sib, coerced):
....:     num = self.numerator()
....:     neg = (num < 0)
....:     if neg: num = -num
....:     if self.denominator()==1:
....:         if coerced:
....:             v = sib.int(num)
....:         else:
....:             pass
....:     else:
....:         return sib(self.numerator())/sib.int(self.denominator())

(continues on next page)
v = sib.name('QQ')(sib.int(num))
else:
v = sib(num)/sib.int(self.denominator())
if neg: v = -v
return v

sage: test_qq_formatter(qq_sage_input_v4)
[-ZZ(5)/7, -ZZ(5)/7, -5/7, -5/7, QQ(3), 3, QQ(3), 3]

AUTHORS:
- Carl Witty (2008-04): new file
- Vincent Delecroix (2015-02): documentation formatting

class sage.misc.sage_input.SIE_assign(sib, lhs, rhs)
Bases: sage.misc.sage_input.SageInputExpression
This class represents an assignment command.

EXAMPLES:

{sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: sib.assign(sib.name('foo').x, sib.name('pi'))
{assign: {getattr: {atomic:foo}.x} {atomic:pi}}

class sage.misc.sage_input.SIE_binary(sib, op, lhs, rhs)
Bases: sage.misc.sage_input.SageInputExpression
This class represents an arithmetic expression with a binary operator and its two arguments, in a sage_input() expression tree.

EXAMPLES:

{sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: sib(3)+5
{binop:+ {atomic:3} {atomic:5}}

class sage.misc.sage_input.SIE_call(sib, func, args, kwargs)
Bases: sage.misc.sage_input.SageInputExpression
This class represents a function-call node in a sage_input() expression tree.

EXAMPLES:

{sage: from sage.misc.sage_input import SageInputBuilder
sage: sie = sib.name('GF')
sage: sie(49)
{call: {atomic:GF}{(atomic:49)}}

class sage.misc.sage_input.SIE_dict(sib, entries)
Bases: sage.misc.sage_input.SageInputExpression
This class represents a dict node in a `sage_input()` expression tree.

EXAMPLES:

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: sib.dict([('\TeX', RR(pi)), ('\Metafont', RR(e))])
```

```python
sage: sib.dict({-40:-40, 0:32, 100:212})
```

```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'])
```

```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').gen()
sage: qq
```

```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
```
class sage.misc.sage_input.SIE_import_name(sib, module, name, alt_name=None)
    Bases: sage.misc.sage_input.SageInputExpression

    This class represents a name which has been imported from a module.

    EXAMPLES:
    
    sage: from sage.misc.sage_input import SageInputBuilder
    sage: sib = SageInputBuilder()
    sage: sib.import_name('sage.rings.integer', 'make_integer')
    {import:sage.rings.integer/make_integer}
    sage: sib.import_name('sage.foo', 'happy', 'sad')
    {import:sage.foo/happy as sad}

    class sage.misc.sage_input.SIE_literal(sib)
    Bases: sage.misc.sage_input.SageInputExpression

    An abstract base class for literals (basically, values which consist of a single token).

    EXAMPLES:
    
    sage: from sage.misc.sage_input import SageInputBuilder, SIE_literal
    sage: sib = SageInputBuilder()
    sage: sie = sib(3)
    sage: sie
    {atomic:3}
    sage: isinstance(sie, SIE_literal)
    True
    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_literal_stringrep(sib, n)
    Bases: sage.misc.sage_input.SIE_literal

    Values in this class are leaves in a sage_input() expression tree. Typically they represent a single token, and consist of the string representation of that token. They are used for integer, floating-point, and string literals, and for name expressions.

    EXAMPLES:
    
    sage: from sage.misc.sage_input import SageInputBuilder, SIE_literal_stringrep
    sage: sib = SageInputBuilder()
    sage: isinstance(sib(3), SIE_literal_stringrep)
    True
    sage: isinstance(sib(3.14159, True), SIE_literal_stringrep)
    True
    sage: isinstance(sib.name('pi'), SIE_literal_stringrep)
    True
    sage: isinstance(sib(False), SIE_literal_stringrep)
    True
    sage: sib(False)
    {atomic:False}

    class sage.misc.sage_input.SIE_subscript(sib, coll, key)
    Bases: sage.misc.sage_input.SageInputExpression

    This class represents a subscript node in a sage_input() expression tree.
EXAMPLES:

```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'] ]}
```

```python
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'})}
```

```python
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}}
```

```python
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\n', 'x/7'); v
x = 22
x/7
sage: isinstance(v, tuple)
True
sage: v[0]
'x = 22\n'
sage: v[1]
'x/7'
sage: len(v)
```

(continues on next page)


```python
sage: v = SageInputAnswer('', 'sin(3.14)', {'sin': math.sin}); v

LOCALS:
    sin: <built-in function sin>
    sin(3.14)

sage: v[0]
''

sage: v[1]
'sin(3.14)'

sage: v[2]
{'sin': <built-in function sin>}
```

```python
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)
    ```

    ```python
    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
    
    sage: sib = SageInputBuilder()
    ```

    ```python
    ```

    ```python
    cache(x, sie, name)
    ```

    INPUT:

    • x - an arbitrary value
```
• **sie** - a *SageInputExpression*

• **name** - a requested variable name

Enters $x$ and **sie** in a cache, so that subsequent calls `self(x)` will directly return **sie**. Also, marks the requested name of this **sie** to be **name**.

This should almost always be called as part of the method `{_sage_input_}` method of a parent. It may also be called on values of an arbitrary type, which may be useful if the values are both large and likely to be used multiple times in a single expression.

**EXAMPLES:**

```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**($v$, **cmd**)

**INPUT:**

• $v$, **cmd** – *SageInputExpression*

Attaches a command to $v$, which will be executed before $v$ is used. Multiple commands will be executed in the order added.

**EXAMPLES:**

```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(**entries**)

Given a dictionary, or a list of (key, value) pairs, produces a *SageInputExpression* representing the dictionary.

**EXAMPLES:**
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[]
y
```

getattr(sie, attr)
Given a SageInputExpression representing foo and an attribute name bar, produce a SageInputExpression representing foo.bar. Normally, you could just use attribute-access syntax, but that doesn't work if bar is some attribute that bypasses __getattr__ (such as if bar is '__getattr__' itself).

```python
```
id_cache\( (x, \text{sie}, \text{name}) \)

INPUT:

- \(x\) - an arbitrary value
- \(\text{sie}\) - a SageInputExpression
- \(\text{name}\) - a requested variable name

Enters \(x\) and \(\text{sie}\) in a cache, so that subsequent calls \(self(x)\) will directly return \(\text{sie}\). Also, marks the requested name of this \(\text{sie}\) to be \(\text{name}\). Differs from the method{cache} method in that the cache is keyed by \(\text{id}(x)\) 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:

```python
sage: from sage.misc.sage_input import SageInputBuilder
sage: x = polygen(ZZ)
sage: sib = SageInputBuilder()
sage: my_42 = 42*x
sage: sie42 = sib(my_42)
sage: sib.id_cache(my_42, sie42, 'the_ultimate_answer')
sage: R.<x> = ZZ[]
the_ultimate_answer = 42*x
the_ultimate_answer + the_ultimate_answer
```

Since id_cache keys off of object identity ("is"), the following does not trigger the cache.: 

```python
sage: sib.result(sib(42*x) + sib(42*x))
42*x + 42*x
```

Note that we don’t assign the result to a variable if the value is only used once.: 

```python
sage: sib = SageInputBuilder()
sage: my_42 = 42*x
sage: sie42 = sib(my_42)
sage: sib.id_cache(my_42, sie42, 'the_ultimate_answer')
sage: R.<x> = ZZ[]
42*x + 43*x
```

import_name\( (\text{module}, \text{name}, \text{alt_name}=None) \)

INPUT:
• module, name, alt_name – strings

Creates an expression that will import a name from a module and then use that name.

EXAMPLES:

```python
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> = QQ[]`).

The method `{'sage_input'}` method of a parent class with generators should construct a SageInputExpression for the parent, and then call this method with the parent itself, the constructed SIE, a sequence containing the names of the generators, and (optionally) another SIE to use if the sage generator syntax is used; typically this will be the same as the first SIE except omitting a names parameter.

**EXAMPLES:**

```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)
```

(continues on next page)
 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 \texttt{Integer} and \texttt{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().prepare()
True
sage: SageInputBuilder(prepare=False).prepare()
False
```

**prod**(factors, simplify=False)

Given a sequence, returns a \texttt{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:**

```python
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:**

```python
sage: from sage.misc.sage_input import SageInputBuilder

Without explicitly using `.share()`, string literals are not shared:

```python
sage: sib = SageInputBuilder()
sage: e = sib('hello')
sage: sib.result(sib((e, e)))
('hello', 'hello')
```

See the difference if we use `.share()`:

```python
sage: sib = SageInputBuilder()
sage: e = sib('hello')
sage: sib.share(e)
sage: sib.result(sib((e, e)))
('hello', 'hello')
```

**sum(terms, simplify=False)**

Given a sequence, returns a *SageInputExpression* for the product of the elements.

With `simplify=True`, performs some simplifications first. If any element is formatted as a string `'0'`, then it is removed from the sequence (unless it is the only element in the sequence); and any instances of `a + -b` are changed to `a - b`.

**EXAMPLES:**

```python
sage: from sage.misc.sage_input import SageInputBuilder

sage: sib = SageInputBuilder()
sage: sib.result(sib.sum([-1, 0, 1, 0, -1]))
```

(continues on next page)
-1 + 0 + 1 + 0 + -1

```python
sage: sib = SageInputBuilder()
sage: sib.result(sib.sum([-1, 0, 1, 0, -1], simplify=True))
-1 + 1 - 1
```

```python
sage: sib = SageInputBuilder()
sage: sib.result(sib.sum([0, 0, 0], simplify=True))
0
```

```python
use_variable(sie, name)
```

Marks the `SageInputExpression` `sie` to use a variable even if it is only referenced once. (If `sie` is the final top-level expression, though, it will not use a variable.)

**EXAMPLES:**

```python
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 `__` 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`. 

300 Chapter 4. Formatted Output
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: sie._sie_prepare(sif)
    sage: sif._commands
    ''
    sage: sif.format(sie, 0)
    'si'
    sage: sif._commands
    'si = matrix(ZZ, 10, 10)\nsi.randomize()\n'
    ```
```
get_name(name)
Return a name corresponding to a given requested name. If only one request for a name is received, then we will use the requested name; otherwise, we will add numbers to the end of the name to make it unique.

If the input name is None, then it is treated as a name of 'si'.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputFormatter
sage: sif = SageInputFormatter()
sage: names = ('x', 'x', 'y', 'z')
sage: for n in names: sif.register_name(n)
sage: for n in names: sif.get_name(n)
'x1'
'x2'
'y'
'z'
```

register_name(name)
Register that some value would like to use a given name. If only one request for a name is received, then we will use the requested name; otherwise, we will add numbers to the end of the name to make it unique.

If the input name is None, then it is treated as a name of 'si'.

EXAMPLES:

```
sage: from sage.misc.sage_input import SageInputFormatter
sage: sif = SageInputFormatter()
sage: sif._names, sif._dup_names
(set(), {})
sage: sif.register_name('x')
sage: sif.register_name('y')
sage: sif._names, sif._dup_names
({'x', 'y'}, {})sage: sif.register_name('x')
sage: sif._names, sif._dup_names
({'x', 'y'}, {'x': 0})
```

sage.misc.sage_input.sage_input(x, preparse=True, verify=False, allow_locals=False)
Return a sequence of commands that can be used to rebuild the object x.

INPUT:

- x - the value we want to find an input form for
- preparse - (default True) Whether to generate code that requires the preparser. With True, generated code requires the preparser. With False, generated code requires that the preparser not be used. With None, generated code will work whether or not the preparser is used.
- verify - (default False) If True, then the answer will be evaluated with sage_eval(), and an exception will be raised if the result is not equal to the original value. (In fact, for verify=True, sage_input() is effectively run three times, with preparse set to True, False, and None, and all three results are checked.) This is particularly useful for doctests.
- allow_locals - (default False) If True, then values that sage_input() cannot handle are returned in a dictionary, and the returned code assumes that this dictionary is passed as the locals parameter of sage_eval(). (Otherwise, if sage_input() cannot handle a value, an exception is raised.)
EXAMPLES:

```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)
('# Verified\n', 'RealField(20)(3.1415939)')
```

We cannot find an input form for a function.:  

```python
sage: sage_input((3, lambda x: x))
Traceback (most recent call last):
...
ValueError: 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)
('# Verified\n', '(3, _sil1)', {'_sil1': <function <lambda> at 0x...>})
```

`sage.misc.sage_input.verify_same(a, b)`

Verify that two Sage values are the same. This is an extended equality test; it checks that the values are equal
and that their parents are equal. (For values which are not Elements, the types are checked instead.)

If the values are the same, we return None; otherwise, we raise an exception.

EXAMPLES:

```python
sage: from sage.misc.sage_input import verify_same
sage: verify_same(1, 1)
sage: verify_same(1, 2)
Traceback (most recent call last):
  ...  
AssertionError: Expected 1 == 2
sage: verify_same(1, 1r)
Traceback (most recent call last):
  ...  
AttributeError: 'int' object has no attribute 'parent'
sage: verify_same(1r, 1)
Traceback (most recent call last):
  ...  
assert(type(a) == type(b))
AssertionError
sage: verify_same(5, GF(7)(5))
Traceback (most recent call last):
  ...  
assert(a.parent() == b.parent())
AssertionError
```

**sage.misc.sage_input.verify_si_answer** *(x, answer, preparse)*

Verify that evaluating `answer` gives a value equal to `x` (with the same parent/type). If `preparse` is True or False, then we evaluate `answer` with the preparser enabled or disabled, respectively; if `preparse` is None, then we evaluate `answer` both with the preparser enabled and disabled and check both results.

On success, we return None; on failure, we raise an exception.

**INPUT:**

- `x` - an arbitrary Sage value
- `answer` - a string, or a `SageInputAnswer`
- `preparse` – True, False, or None

**EXAMPLES:**

```python
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):
  ...  
AssertionError: '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 `table` for details and examples.

AUTHORS:

- John H. Palmieri (2012-11)

```python
class sage.misc.table.table(rows=None, columns=None, header_row=False, header_column=False, frame=False, align='left')

Bases: sage.structure.sage_object.SageObject

Display a rectangular array as a table, either in plain text, LaTeX, or html.

INPUT:

- rows (default None) - a list of lists (or list of tuples, etc.), containing the data to be displayed.
- columns (default None) - a list of lists (etc.), containing the data to be displayed, but stored as columns. Set either rows or columns, but not both.
- header_row (default False) - if True, first row is highlighted.
- header_column (default False) - if True, first column is highlighted.
- frame (default False) - if True, put a box around each cell.
- align (default ‘left’) - the alignment of each entry: either ‘left’, ‘center’, or ‘right’

EXAMPLES:

```
sage: rows = [['a', 'b', 'c'], [100, 2, 3], [4, 5, 60]]
sage: table(rows)
a b c
100 2 3
4 5 60
```

If `header_row` is True, then the first row is highlighted. If `header_column` is True, then the first column is highlighted. If `frame` is True, then print a box around every “cell”.

```python
sage: table(rows, header_row=True)
 a b c
+-------+-----+
|100    | 2    |
| 4     | 5    |

sage: latex(table(rows, header_row=True))
\begin{tabular}{lll}
a & b & c \\ 
$100$ & $2$ & $3$ \\
$4$ & $5$ & $60$ \\
\end{tabular}
```

```python
sage: table(rows=rows, frame=True)
```

(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:

\begin{verbatim}
sage: table(columns=[[x,n(sin(x), digits=2)) for x in [0..3]], header_column=['$x$', r'$\sin(x)$'], frame=True)
\end{verbatim}

\begin{tabular}{l|llll}
\hline
\hline
$x$ & 0 & 1 & 2 & 3 \\
\hline
\hline
$\sin(x)$ & 0.00 & 0.84 & 0.91 & 0.14 \\
\hline
\end{tabular}

or by taking the \texttt{transpose()} of the original table:

\begin{verbatim}
sage: table(rows=[[x,n(sin(x), digits=2)) for x in [0..3]], header_row=['$x$', r'$\sin(x)$'], frame=True).transpose()
\end{verbatim}

\begin{tabular}{l|llll}
\hline
\hline
$x$ & 0 & 1 & 2 & 3 \\
\hline
\hline
$\sin(x)$ & 0.00 & 0.84 & 0.91 & 0.14 \\
\hline
\end{tabular}

In either plain text or LaTeX, entries in tables can be aligned to the left (default), center, or right:

\begin{verbatim}
sage: table(rows, align='left')
a b c
100 2 3
4 5 60
sage: table(rows, align='center')
a b c
100 2 3
4 5 60
sage: table(rows, align='right', frame=True)
+-------++
| a | b | c |
+-------++
| 100 | 2 | 3 |
+-------++
| 4 | 5 | 60 |
+-------++
\end{verbatim}

To generate HTML you should use \texttt{html(table(...))}:

\begin{verbatim}
sage: data = [['$x$'], ['$\sin(x)$']] + [[x,n(sin(x), digits=2)) for x in [0..3]]
sage: output = html(table(data, header_row=True, frame=True))
sage: type(output)
<class 'sage.misc.html.HtmlFragment'>
\end{verbatim}

(continues on next page)
It is an error to specify both rows and columns:

```sage
table(rows=[[1,2,3], [4,5,6]], columns=[[0,0,0], [0,0,1024]])
```
Traceback (most recent call last):
...  
ValueError: Don't set both 'rows' and 'columns' when defining a table.

```sage
table(columns=[[0,0,0], [0,0,1024]])
```
0 0 0
0 0 1024

Note that if rows is just a list or tuple, not nested, then it is treated as a single row:

```sage
table([1,2,3])
```
1 2 3

Also, if you pass a non-rectangular array, the longer rows or columns get truncated:

```sage
table([[1,2,3,7,12], [4,5]], 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:

```python
sage: from sage.repl.rich_output import get_display_manager
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:

```python
sage: T = table([['a', 'b', 'c'], [1, 2, 3]])
sage: T.options()['align'], T.options()['frame'] ('left', False)
sage: T.options(align='right', frame=True)
sage: T.options()['align'], T.options()['frame'] ('right', True)
```

Note that when first initializing a table, header_row or header_column can be a list. In this case, during
the initialization process, the header is merged with the rest of the data, so changing the header option later
using table.options(...) doesn't affect the contents of the table, just whether the row or column is
highlighted. When using this options() method, no merging of data occurs, so here header_row and
header_column should just be True or False, not a list.

```python
sage: T = table([[1,2,3], [4,5,6]], header_row=['a', 'b', 'c'], frame=True)
sage: T
+---+---+---+
| 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.
```

**transpose()**

Return a table which is the transpose of this one: rows and columns have been interchanged. Several of the properties of the original table are preserved: whether a frame is present and any alignment setting. On the other hand, header rows are converted to header columns, and vice versa.

**EXAMPLES:**

```python
sage: T = table([[1,2,3], [4,5,6]])
sage: T.transpose()
1 4
2 5
3 6

sage: T = table([[1,2,3], [4,5,6]], header_row=['x', 'y', 'z'], frame=True)
sage: T.transpose()
+-----+-----+-----+
| x   | 1   | 4   |
| y   | 2   | 5   |
| z   | 3   | 6   |
+-----+-----+-----+
```

## 4.2 HTML and MathML

### 4.2.1 HTML Fragments

This module defines a HTML fragment class, which holds a piece of HTML. This is primarily used in browser-based notebooks, though it might be useful for creating static pages as well.

This module defines **MathJax**, an object of which performs the task of producing an HTML representation of any object. The produced HTML is renderable in a browser-based notebook with the help of MathJax.

```python
class sage.misc.html.HTMLFragmentFactory
    Bases: sage.structure.sage_object.SageObject

    eval(s, locals=None)
        Evaluate embedded <sage> tags

    INPUT:
        * s – string.
```
• `globals` – dictionary. The global variables when evaluating `s`. Default: the current global variables.

OUTPUT:
A `HtmlFragment` instance.

EXAMPLES:

```sage
a = 123
html.eval('<sage>a</sage>')
(123)
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>
pretty_print(html.iframe("http://sagemath.org",30,40))
<iframe height="30" width="40"
src="http://sagemath.org"></iframe>
pretty_print(html.iframe("https://sagemath.org",30))
<iframe height="30" width="800"
src="https://sagemath.org"></iframe>
pretty_print(html.iframe("/home/admin/0/data/filename"))
<iframe height="400" width="800"
src="/home/admin/0/data/filename"></iframe>
pretty_print(html.iframe('data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAUAAAAFCAYAAACNbyblAAAAHElEQVQI12P4//8/w38GIAXDIBKE0DHxgljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg="'))
<iframe height="400" width="800"
src="http://data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAUAAAAFCAYAAACNbyblAAAAHElEQVQI12P4//8/w38GIAXDIBKE0DHxgljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg=""></iframe>
```

### `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: from sage.misc.html import HtmlFragment
sage: HtmlFragment('&lt;b&gt;test&lt;/b&gt;')
&lt;b&gt;test&lt;/b&gt;

_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
dm = get_display_manager()
h = sage.misc.html.HtmlFragment('&lt;b&gt;old&lt;/b&gt;')
h._rich_repr_(dm)  # the doctest backend does not support html

class sage.misc.html.MathJax

Bases: object

Render LaTeX input using MathJax. This returns a MathJaxExpr.

EXAMPLES:

sage: from sage.misc.html import MathJax
MathJax()(3)
<html>
\[
\newcommand{\Bold}[1]{\mathbf{#1}}3\]
</html>

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
MathJax().eval(3, mode='display')
<html>
\[
\newcommand{\Bold}[1]{\mathbf{#1}}\mathbf{3}\]
</html>

MathJax().eval(3, mode='inline')
<html>
\[
\newcommand{\Bold}[1]{\mathbf{#1}}\Bold{3}\]
</html>

MathJax().eval(type(3), mode='inline')
<html>
\[
\newcommand{\Bold}[1]{\mathbf{#1}}\verb|&lt;class|\verb|sage.rings.integer.Integer|\verb|&gt;|\]
</html>
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\[\text\]{\textbf{#1}}\frac{1}{2}\]
</html>
sage: html('<a href="http://sagemath.org">sagemath</a>')
<a href="http://sagemath.org">sagemath</a>
sage: html('<a href="http://sagemath.org">sagemath</a>', strict=True)
<a href="http://sagemath.org">sagemath</a>
```

sage.misc.html.math_parse(s)

Transform the string `s` with TeX maths to an HTML string renderable by MathJax.

INPUT:

- `s` – a string

OUTPUT:

A `HtmlFragment` instance.

Specifically this method does the following:

- Replace all `$text$'s by `\(text\)`
- Replace all `$$text$$'s by `\[text\]`
• Replace all $’s by $’s. Note that this has precedence over the above two cases.

EXAMPLES:

```
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>$</span>2+2<span>$</span> is rendered to \(2+2\).
```

```
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:

```
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.

```
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 sage.misc.html.

AUTHORS:

- William Stein: original implementation
- Joel B. Mohler: latex_variable_name() drastic rewrite and many doc-tests

```python
class sage.misc.latex.Latex(debug=False, slide=False, density=150, pdflatex=None, engine=None)
    Bases: sage.misc.latex.LatexCall
```

Enter, e.g.,

```latex
The equation $y^2 = x^3 + x$ defines an elliptic curve.
We have $2006 = \sage{factor(2006)}$.
```

in an input cell in the notebook to get a typeset version. Use \%latex\_\_debug to get debugging output.

Use latex(...) to typeset a Sage object. Use LatexExpr to typeset LaTeX code that you create by hand.

Use \%slide instead to typeset slides.

**Warning:** You must have dvipng (or dvips and convert) installed on your operating system, or this command will not work.

EXAMPLES:

```python
sage: latex(x^20 + 1)
x^{20} + 1
sage: latex(FiniteField(25, 'a'))
\Bold{F}_{5^{2}}
sage: latex("hello")
\text{hello}
sage: LatexExpr(r"\frac{x^2 - 1}{x + 1} = x - 1")
\frac{x^2 - 1}{x + 1} = x - 1
```

LaTeX expressions can be added; note that a space is automatically inserted:
\texttt{sage: LatexExpr(r"y \neq") + latex(x^20 + 1)}

\texttt{y \neq x^{20} + 1}

\textbf{add\_macro}(\textit{macro})

Append to the string of extra LaTeX macros, for use with \texttt{\%latex} and \texttt{\%html}.

INPUT:

- \textit{macro} – string

EXAMPLES:

\begin{verbatim}
sage: latex.extra_macros()
''
sage: latex.add_macro("\\newcommand{\foo}{bar}")
sage: latex.extra_macros()
'\\newcommand{\foo}{bar}'
sage: latex.extra_macros('')  # restore to default
\end{verbatim}

\textbf{add\_package\_to\_preamble\_if\_available}(\textit{package\_name})

Adds a \texttt{\usepackage{package\_name}} instruction to the latex preamble if not yet present there, and if \texttt{package\_name.sty} is available in the LaTeX installation.

INPUT:

- \textit{package\_name} – a string

See also:

- \texttt{add\_to\_preamble()}
- \texttt{has\_file()}.

\textbf{add\_to\_preamble}(\textit{s})

Append to the string \textit{s} of extra LaTeX macros, for use with \texttt{\%latex}.

EXAMPLES:

\begin{verbatim}
sage: latex.extra_preamble()
''
sage: latex.add_to_preamble("\\DeclareMathOperator{\Ext}{Ext}\usepackage{xypic}\"")
\end{verbatim}

At this point, a notebook cell containing

\begin{verbatim}
\%latex
\$\Ext_A^*(\GF{2}, \GF{2}) \Rightarrow \pi_*^{s^s}(S^0)\$
\end{verbatim}

will be typeset correctly.

\begin{verbatim}
sage: latex.add_to_preamble("\\usepackage{xypic}\")
sage: latex.extra_preamble()
'\\DeclareMathOperator{\Ext}{Ext}\\usepackage{xypic}'
\end{verbatim}

Now one can put various xypic diagrams into a \texttt{\%latex} cell, such as

\begin{verbatim}
\%latex
\[
\xymatrix{ \circ \ar[r]\ar[d]^a & \circ\ar[r]^b \ar[d]^c & \circ\ar[r]^{d}\ar[d]^e & \circ\ar[r]^f & \circ}
\]
\end{verbatim}

(continues on next page)
Reset the preamble to its default, the empty string:

```
sage: latex.extra_preamble('')
sage: latex.extra_preamble()
''
```

blackboard_bold(t=None)

Controls whether Sage uses blackboard bold or ordinary bold face for typesetting $\mathbb{Z}$, $\mathbb{R}$, etc.

**INPUT:**

- t – boolean or None

**OUTPUT:**

If t is None, return the current setting (True or False).

If t is True, use blackboard bold ($\mathbb{Z}$); otherwise use boldface ($\mathbf{Z}$).

**EXAMPLES:**

```
sage: latex.blackboard_bold()
False
sage: latex.blackboard_bold(True)
True
sage: latex.blackboard_bold(False)
```

check_file(file_name, more_info='')

**INPUT:**

- file_name – a string
- more_info – a string (default: '')

Emit a warning if the local LaTeX installation does not include file_name. The string more_info is appended to the warning message. The warning is only emitted the first time this method is called.

**EXAMPLES:**

```
sage: latex.check_file("article.cls")  # optional - latex
sage: latex.check_file("some_inexistent_file.sty")
Warning: `some_inexistent_file.sty` is not part of this computer's TeX\text{} installation.
sage: latex.check_file("some_inexistent_file.sty")
Warning: `some_inexistent_file.sty` is not part of this computer's TeX\text{} installation.
This file is required for blah. It can be downloaded from: http://blah.org/
```

This test checks that the bug in trac ticket #9091 is fixed:

```
sage: latex.check_file("article.cls", "The article class is really critical.")  # optional - latex
```
**engine**(*e=None*)

Set Sage to use *e* as latex engine when typesetting with *view()*, in %latex cells, etc.

**INPUT:**

- *e* – ‘latex’, ‘pdflatex’, ‘xelatex’ or *None*

  If *e* is *None*, return the current engine.

  If using the XeLaTeX engine, it will almost always be necessary to set the proper preamble with *extra_preamble()* or *add_to_preamble()*.

  For example:

  ```python
  latex.extra_preamble(r"
  \usepackage{fontspec,xunicode,xltxtra}
  \setmainfont[Mapping=tex-text]{some font here}
  \setmonofont[Mapping=tex-text]{another font here}"
  )
  ```

**EXAMPLES:**

```
sage: latex.engine()
'pdflatex'
sage: latex.engine("latex")
sage: latex.engine()
'latex'
sage: latex.engine("xelatex")
sage: latex.engine()
'xelatex'
```

**eval**(*x, globals=False, strip=False, filename=None, debug=None, density=None, pdflatex=None, engine=None, locals={}*)

Compile the formatted tex given by *x* as a png and writes the output file to the directory given by *filename*.

**INPUT:**

- *globals* – a globals dictionary
- *x* – string to evaluate.
- *strip* – ignored
- *filename* – output filename
- *debug* – whether to print verbose debugging output
- *density* – how big output image is.
- *pdflatex* – whether to use pdflatex. This is deprecated. Use *engine* option instead.
- *engine* – latex engine to use. Currently latex, pdflatex, and xelatex are supported.
- *locals* - extra local variables used when evaluating Sage code in *x*.

**Warning:** When using latex (the default), you must have ‘dvipng’ (or ‘dvips’ and ‘convert’) installed on your operating system, or this command will not work. When using pdflatex or xelatex, you must have ‘convert’ installed.

**OUTPUT:**

If it compiled successfully, this returns an empty string ‘’, otherwise it returns *None*.

**EXAMPLES:**
sage: fn = tmp_filename()
sage: latex.eval("$\ZZ[x]$", locals(), filename=fn) # not tested
''
sage: latex.eval(r"\ThisIsAnInvalidCommand", {}) # optional -- latex ImageMagick
An error occurred...
No pages of output...

extra_macros(macro=None)
String containing extra LaTeX macros to use with %latex and %html.

INPUT:

• macro – string (default: None)

If macro is None, return the current string. Otherwise, set it to macro. If you want to append to the string of macros instead of replacing it, use latex.add_macro.

EXAMPLES:

sage: latex.extra_macros("\newcommand{\foo}{bar}")
sage: latex.extra_macros()  #latex.add_macro
\newcommand{\foo}{bar}

extra_preamble(s=None)
String containing extra preamble to be used with %latex.

INPUT:

• s – string or None

If s is None, return the current preamble. Otherwise, set it to s. If you want to append to the current extra preamble instead of replacing it, use latex.add_to_preamble.

You will almost certainly need to use this when using the XeLaTeX engine; see below or the documentation for engine() for a suggested preamble.

EXAMPLES:

sage: latex.extra_preamble("\DeclareMathOperator{\Ext}{Ext}")
sage: latex.extra_preamble()  #latex.add_to_preamble
\DeclareMathOperator{\Ext}{Ext}

has_file(file_name)
INPUT:

• file_name – a string
Tests whether the local LaTeX installation includes `file_name`.

**EXAMPLES:**

```python
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:**

```python
sage: a = matrix(1, 1, [42])
sage: latex(a)
\left(\begin{array}{r}
42
\end{array}\right)
sage: latex.matrix_column_alignment('c')
sage: latex(a)
\left(\begin{array}{c}
42
\end{array}\right)
sage: latex.matrix_column_alignment('l')
sage: latex(a)
\left(\begin{array}{l}
42
\end{array}\right)
```

Restore defaults:

```python
sage: latex.matrix_column_alignment('r')
```

### matrix_delimiters(left=None, right=None)

Change the left and right delimiters for the LaTeX representation of matrices.

**INPUT:**

- **left, right** - strings or `None`.

If both `left` and `right` are `None`, then return the current delimiters. Otherwise, set the left and/or right delimiters, whichever are specified.

Good choices for `left` and `right` are any delimiters which LaTeX understands and knows how to resize; some examples are:

- parentheses: `(, )`
- brackets: `[`, `]`
Note: Putting aside aesthetics, you may combine these in any way imaginable; for example, you could set \texttt{left} to be a right-hand bracket ‘]’ and \texttt{right} to be a right-hand brace ‘\}’, and it will be typeset correctly.

\textbf{EXAMPLES:}

\begin{Verbatim}
\begin{verbatim}
sage: a = matrix(1, 1, [17])
sage: latex(a) \end{verbatim}
17
\end{verbatim}
\begin{verbatim}
sage: \texttt{latex.matrix_delimiters(“[”, “]“)}
sage: latex(a) \end{verbatim}
17
\end{verbatim}
\begin{verbatim}
sage: \texttt{latex.matrix_delimiters(left=”{“)}
sage: latex(a) \end{verbatim}

Restore defaults:

\begin{verbatim}
sage: \texttt{latex.matrix_delimiters(“(”, “)”)}
\end{verbatim}

\textbf{vector \_delimiters}(left=None, right=None)

Change the left and right delimiters for the LaTeX representation of vectors.

INPUT:

- \texttt{left, right} – strings or \texttt{None}

If both \texttt{left} and \texttt{right} are \texttt{None}, then return the current delimiters. Otherwise, set the left and/or right delimiters, whichever are specified.

Good choices for \texttt{left} and \texttt{right} are any delimiters which LaTeX understands and knows how to resize; some examples are:

- parentheses: ‘(, ’)
- brackets: ‘[, ]’
- braces: ‘\{, \}’
- vertical lines: ‘|’
- angle brackets: ‘\langle, \rangle’

Note: Putting aside aesthetics, you may combine these in any way imaginable; for example, you could set
left to be a right-hand bracket ‘]’ and right to be a right-hand brace ‘\}', and it will be typeset correctly.

EXAMPLES:

```
sage: a = vector(QQ, [1,2,3])
sage: latex(a)
\left(1,\,2,\,3\right)
sage: latex(vector_delimiters("[", "]")
\left[1,\,2,\,3\right]
sage: latex(vector_delimiters(right="\}")
\left[1,\,2,\,3\right]
sage: latex(vector_delimiters())
\[''', '\\}'']
```

Restore defaults:

```
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:

```
sage: from sage.misc.latex import LatexCall
sage: LatexCall()(ZZ)
\Bold{Z}
sage: LatexCall().__call__(ZZ)
\Bold{Z}
```

This returns an instance of the class LatexExpr:

```
sage: type(LatexCall()(ZZ))
<class 'sage.misc.latex.LatexExpr'>
```

class sage.misc.latex.LatexExamples

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:

```
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 `latex()` call. It is also possible to generate `LatexExpr` directly from a string, which must contain valid LaTeX code for typesetting in math mode (without dollar signs). In the Sage notebook, use `pretty_print()` or the “Typeset” checkbox to actually see the typeset LaTeX code; alternatively, from either the command-line or the notebook, use the `view()` function.

**INPUT:**

- `str` – a string with valid math mode LaTeX code (or something which can be converted to such a string).

**OUTPUT:**

- `LatexExpr` wrapping the string representation of the input.

**EXAMPLES:**

```
sage: latex(x^20 + 1)
x^{20} + 1
sage: LatexExpr(r'\frac{x^2 + 1}{x - 2}')
\frac{x^2 + 1}{x - 2}
```

`LatexExpr` simply converts to string without doing anything extra, it does not call `latex()`:

```
sage: latex(ZZ)
\Bold{Z}
sage: LatexExpr(ZZ)
Integer Ring
```

The result of `latex()` is of type `LatexExpr`:

```
sage: L = latex(x^20 + 1)
sage: L
x^{20} + 1
sage: type(L)
<class 'sage.misc.latex.LatexExpr'>
```

A `LatexExpr` can be converted to a plain string:

```
sage: str(latex(x^20 + 1))
'x^{20} + 1'
```

`sage.misc.latex.None_function(x)`

Returns the LaTeX code for `None`.

**INPUT:**

- `x` – `None`

**EXAMPLES:**

```
sage: from sage.misc.latex import None_function
sage: print(None_function(None))
\mathrm{None}
```

`sage.misc.latex.bool_function(x)`

Returns the LaTeX code for a boolean `x`.

**INPUT:**

- `x` – boolean
EXCEPTIONS:

```
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:**

```
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:**

```
sage: from sage.misc.latex import coeff_repr
sage: coeff_repr(QQ(1/2))
'\frac{1}{2}'
sage: coeff_repr(-x^2)
'(-x^2)'
```

`sage.misc.latex.dict_function(x)`

Return the LaTeX code for a dictionary `x`.

**INPUT:**

- `x` – a dictionary

**EXAMPLES:**
sage: from sage.misc.latex import dict_function
sage: x,y,z = var('x,y,z')
sage: print(dict_function({x/2: y^2}))
\left\{\frac{1}{2} \, x : y^{2}\right\}
sage: d = {(1,2,x^2): [sin(z^2), y/2]}
sage: latex(d)
\left\{\left(1, 2, x^{2}\right) : \left[\sin\left(z^{2}\right), \frac{1}{2} \, y\right]\right\}

```
sage.misc.latex.float_function(x)
Returns the LaTeX code for a python float x.

INPUT:
• x – a python float

EXAMPLES:
```
sage: from sage.misc.latex import float_function
sage: float_function(float(3.14))
3.14
sage: float_function(float(1e-10))
1 \times 10^{-10}
sage: float_function(float(2e10))
2000000000.0

```
sage.misc.latex.has_latex_attr(x)
Return True if x has a _latex_ attribute, except if x is a type, in which case return False.

EXAMPLES:
```
sage: from sage.misc.latex import has_latex_attr
sage: has_latex_attr(identity_matrix(3))
True
sage: has_latex_attr("abc")  # strings have no _latex_ method
False
```

Types inherit the _latex_ method of the class to which they refer, but calling it is broken:
```
sage: T = type(identity_matrix(3)); T
<class '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:

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:

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:

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:

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:

\begin{verbatim}
  sage: latex(Integer(3))  # indirect doctest
  3
  sage: latex(1==0)
  \text{False}
  sage: print(latex([x,2]))
  [x, 2]
\end{verbatim}

Check that trac ticket #11775 is fixed:

\begin{verbatim}
  sage: latex((x,2), combine_all=True)
  x 2
\end{verbatim}

\section*{sage.misc.latex.latex_extra_preamble()}

Return the string containing the user-configured preamble, \texttt{sage_latex_macros}, and any user-configured macros. This is used in the \texttt{eval()} method for the \texttt{Latex} class, and in \texttt{_latex_file()}; it follows either \texttt{LATEX_HEADER} or \texttt{SLIDE_HEADER} (defined at the top of this file) which is a string containing the documentclass and standard \usepackage commands.

\textbf{EXAMPLES:}

\begin{verbatim}
  sage: from sage.misc.latex import latex_extra_preamble
  sage: print(latex_extra_preamble())
  \newcommand{\ZZ}{\Bold{\mathbb{Z}}}
  \newcommand{\NN}{\Bold{\mathbb{N}}}
  \newcommand{\RR}{\Bold{\mathbb{R}}}
  \newcommand{\CC}{\Bold{\mathbb{C}}}
  \newcommand{\QQ}{\Bold{\mathbb{Q}}}
  \newcommand{\QQbar}{\overline{\QQ}}
  \newcommand{\GF}{\Bold{\mathbb{F}}}
  \newcommand{\Zp}{\Bold{\mathbb{Z}}}
  \newcommand{\Qp}{\Bold{\mathbb{Q}}}
  \newcommand{\Zmod}{\ZZ/#1\ZZ}
  \newcommand{\CDF}{\Bold{\mathbb{C}}}
  \newcommand{\CIF}{\Bold{\mathbb{I} \mathbb{R}}}
  \newcommand{\RLF}{\Bold{\mathbb{R}}}
  \newcommand{\Bold}{\mathbf{#1}}
\end{verbatim}

\section*{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('sigma_{389}')
'\sigma_{389}'
sage: latex_variable_name('beta_{00}')
'\beta_{00}'
sage: latex_variable_name('Omega_{84}')
'\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)
'\text{nothing}_{1}'
sage: latex_variable_name('nothing_{abc}')
'\text{nothing}_{abc}'
sage: latex_variable_name('nothing_{abc}', is_fname=True)
'\text{nothing}_{abc}'
sage: latex_variable_name('alpha_{beta_{gamma_{12}}}')
'\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 “\text{a}” or “\text{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')
'\text{aleph}'
sage: latex_varify('aleph', is_fname=True)
'\text{aleph}'
sage: latex_varify('alpha')
'\alpha'
```

(continues on next page)
'\alpha'
sage: latex_varify('ast')
'\ast'

sage.misc.latex.list_function(x)
Returns the \LaTeX{} code for a list \texttt{x}.

INPUT: \texttt{x} - a list

EXAMPLES:

\begin{verbatim}
sage: \textbf{from sage.misc.latex import} list_function
sage: \textbf{list_function}([1, 2, 3])
'\left[1, 2, 3\right]'
sage: \textbf{latex}(([1, 2, 3])) \ # indirect doctest
\left[1, 2, 3\right]
sage: \textbf{latex}([[\text{Matrix}(\text{ZZ}, 3, \text{range}(9)), \text{Matrix}(\text{ZZ}, 3, \text{range}(9))]]) \ # indirect doctest
\left[\begin{array}{rrr}
0 & 1 & 2 \\
3 & 4 & 5 \\
6 & 7 & 8
\end{array}\right], \left[\begin{array}{rrr}
0 & 1 & 2 \\
3 & 4 & 5 \\
6 & 7 & 8
\end{array}\right]
\end{verbatim}

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 \texttt{x} and save to the given filename.

INPUT:
  • \texttt{x} – object to be displayed
  • \texttt{filename} – file in which to save the image
  • \texttt{density} – integer (default: 150)
  • \texttt{debug} – bool (default: False): print verbose output
  • \texttt{do_in_background} – bool (default: False): Unused, kept for backwards compatibility
  • \texttt{tiny} – bool (default: False): use ‘tiny’ font
  • \texttt{pdflatex} – bool (default: True): use pdflatex. This option is deprecated. Use \texttt{engine} option instead. See below.
  • \texttt{engine} – (default: 'pdflatex') 'latex', 'pdflatex', or 'xelatex'

EXAMPLES:

\begin{verbatim}
sage: \textbf{from sage.misc.latex import} png
sage: \textbf{png}([ZZ[x]], os.path.join(SAGE_TMP, "zz.png")) \ # random, optional - latex...
\end{verbatim}

sage.misc.latex.repr_lincomb(symbols, coeffs)
Compute a \LaTeX{} representation of a linear combination of some formal symbols.

INPUT:
• symbols – list of symbols
• coeffs – list of coefficients of the symbols

OUTPUT:
A string

EXAMPLES:

```
sage: t = PolynomialRing(QQ, 't').0
sage: from sage.misc.latex import repr_lincomb
sage: repr_lincomb(['a', 's', 't'], [-t, t - 2, t^12 + 2])
'-t\text{\texttt{a}} + \left(t - 2\right)\text{\texttt{s}} + \left(t^{12} + \rightarrow 2\right)\texttt{t}'
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{\texttt{hello\_world}}'
sage: str_function('-1.00000?') # trac 12178
'-1.00000?'
```

declare function to convert a string to LaTeX

```python
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'
```

```python
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
• sep – string (default: ''): separator between math objects
• tiny – bool (default: False): use tiny font.
• pdflatex – bool (default: False): use pdflatex. This is deprecated. Use 'engine' option instead.
• engine – string or None (default: None). Can take the following values:
  – None – the value defined in the LaTeX global preferences latex.engine() is used.
  – 'pdflatex' – compilation does tex -> pdf
  – 'xelatex' – compilation does tex -> pdf
  – 'latex' – compilation first tries tex -> dvi -> png and if an error occurs then tries dvi -> ps -> pdf.
    This is slower than 'pdflatex' and known to be broken when overfull hbox are detected.
• viewer – string or None (default: None): specify a viewer to use; currently the only options are None and 'pdf'.
• tightpage – bool (default: True): use the LaTeX package ‘preview’ with the ‘tightpage’ option.
• margin – float or None (default: None): adds a margin of margin mm; has no affect if the option tightpage is False.
• mode – string (default: 'inline'): 'display' for displaymath or 'inline' for inline math
• combine_all – bool (default: False): If combine_all is True and the input is a tuple, then it does not return a tuple and instead returns a string with all the elements separated by a single space.

OUTPUT:
Display typeset objects.

The output is displayed in a separate viewer displaying a dvi (or pdf) file, with the following: the title string is printed, centered, at the top. Beneath that, each object in objects is typeset on its own line, with the string sep inserted between these lines.

The value of sep is inserted between each element of the list objects; you can, for example, add vertical space between objects with sep='\vspace{15mm}', while sep='\hrule' adds a horizontal line between objects, and sep='\newpage' inserts a page break between objects.

If pdflatex is True, then the latex engine is set to pdflatex.

If the engine is either pdflatex or xelatex, it produces a pdf file. Otherwise, it produces a dvi file, and if the program dvipng is installed, it checks the dvi file by trying to convert it to a png file. If this conversion fails, the dvi file probably contains some postscript special commands or it has other issues which might make displaying it a problem; in this case, the file is converted to a pdf file, which is then displayed.

Setting viewer to 'pdf' forces the use of a separate viewer, even in notebook mode. This also sets the latex engine to be pdflatex if the current engine is latex.

Setting the option tightpage to True (this is the default setting) tells LaTeX to use the package ‘preview’ with the ‘tightpage’ option. Then, each object is typeset in its own page, and that page is cropped to exactly the size of the object. This is typically useful for very large pictures (like graphs) generated with tikz. This only works when using a separate viewer. Note that the object are currently typeset in plain math mode rather than displaymath, because the latter imposes a limit on the width of the picture. Technically, tightpage adds

\usepackage[tightpage,active]{preview}
\PreviewEnvironment{page}


4.3. LaTeX
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.

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
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:
from sage.misc.latex_macros import produce_latex_macro
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:

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:

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}"', ...]
SAVING AND LOADING SAGE OBJECTS

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 pickles that have a better chance of being read on Python 2, by using protocol version 2 (instead of 4) and fixing up imports of standard library modules and types whose names changed between Python 2 and 3. This is enabled by default for the best chances of cross-Python compatibility.
- Further arguments are passed to `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 = table[obj_id]
    return table[obj_id]
```

```python
classmethod dumps(obj, **kwargs)
```

Equivalent to `pickle.dumps()` but using the `sage.misc.persist.SagePickler`.

**INPUT:**

- `obj` - the object to pickle.
- `**kwargs` - keyword arguments passed to the `sage.misc.persist.SagePickler` constructor.

**OUTPUT:**
• pickle - the pickled object as bytes.

EXAMPLES:

```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:

5.2. Object persistence 339
• **data** - the pickle data as bytes.
• **kwargs** - keyword arguments passed to the `sage.misc.persist.SageUnpickler` constructor.

**OUTPUT:**
• **obj** - the object that was serialized to the given pickle data.

**EXAMPLES:**

```python
sage: import pickle
sage: from sage.misc.persist import SageUnpickler
sage: gherkin = pickle.dumps(1)
sage: SageUnpickler.loads(gherkin)
1
```

### sage.misc.persist.db(name)

Load object with given name from the Sage database. Use `x.db(name)` or `db_save(x, name)` to save objects to the database.

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

### sage.misc.persist.db_save(x, name=None)

Save `x` to the Sage database.

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

### sage.misc.persist.dumps(obj, compress=True)

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

See also:

`loads()`

**EXAMPLES:**

```python
sage: a = 2/3
sage: s = dumps(a)
sage: a2 = loads(s)
sage: type(a) is type(a2)
True
sage: a2
2/3
```

### sage.misc.persist.load(compress=True, verbose=True, *filename, **kwargs)

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

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

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

When a pickle created with Python 2 is unpickled in Python 3, Sage uses the default encoding `latin1` to unpickle data of type `str`.

Finally, if you give multiple positional input arguments, then all of those files are loaded, or all of the objects are loaded and a list of the corresponding loaded objects is returned.
If `compress` is true (the default), then the data stored in the file are supposed to be compressed. If `verbose` is true (the default), then some logging is printed when accessing remote files. Further keyword arguments are passed to `pickle.load()`.

**EXAMPLES:**

```python
sage: u = 'https://www.sagemath.org/files/test.sobj'
sage: s = load(u)  # optional - internet
Attempting to load remote file: https://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
'''
sage: t = tmp_filename(ext=".F")
sage: with open(t, 'w') as f:
    ....:    _ = f.write(code)
sage: load(t)
sage: hello
<fortran object>
```

Sage uses `sage.misc.persist.load_sage_element(cls, parent, dic_pic)` and `sage.misc.persist.load_sage_object(cls, dic)` for persistence.

---

**5.2. Object persistence** 341
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:

```
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().

```
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.

```
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\x07''')) in Python-2.

```
sage: g = loads(b"x'\x9c\x80\x07\x80\x9c\x8f\xe7'd'\x0cej`/\r
\n\n\nczą\xc5\xe4\xe7\xad\xc1\xc5\xe7\x89\x87\x89\x8f\x8f\x86\x8b\x8e\x8f\x8f\x8f\x8f\xc1\x87\x80\x80\x80\x80\x80\x80''')
sage: type(g), g.bar
<class '__main__.Foo'>, '\x80\x07'
```

The following line demonstrates what would happen without trac ticket #28444:

```
sage: loads(b'x\x9c'\x8e\x8f\x8f\x8f\x8f\x8f\xc1\x87\x80\x80\x80\x80\x80\x80''', encoding='ASCII')
Traceback (most recent call last):
  ...^
UnicodeDecodeError: 'ascii' codec can't decode byte 0x80 in position 0: ordinal not in range(128)
```

(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:

```
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:

```
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:

```
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:

```
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')
```

5.2. Object persistence
In many cases, unpickling problems for old pickles can be resolved with a simple call to :func:`register_unpickle_override`, as in the example above and in many of the :class:`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 :meth:`__setstate__()` method.

```
object.__setstate__(state)

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

By implementing a :meth:`__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 :class:`CombinatorialObject` object using a class which also inherits from :class:`Element`. This exact problem often arises when refactoring old code into the element framework. First we create a pickle to play with:

```
sage: from sage.structure.element import Element
sage: class SourPickle(CombinatorialObject): pass
sage: class SweetPickle(CombinatorialObject, Element): pass
sage: import __main__
sage: __main__.SourPickle = SourPickle
sage: __main__.SweetPickle = SweetPickle  # a hack to allow us to pickle command-line classes
sage: gherkin = dumps(SourPickle([1, 2, 3]))
```

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

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

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

```
sage: class SweeterPickle(CombinatorialObject, Element):
    ....:     def __setstate__(self, state):
    ....:         # a pickle from CombinatorialObject is just its instance
    ....:         # dictionary
    ....:         if isinstance(state, dict):
    ....:             # this is a fudge: we need an appropriate parent here
    ....:             self._set_parent(Tableaux())
    ....:             self.__dict__ = state
    ....:     else:
    ....:         P, D = state
```

(continues on next page)
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:
    ....:            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')})

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
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\_](\_\_0[\_\_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:**

```python
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',
→ '.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 unpickle_override, register_unpickle_override
sage: unpickle_global('sage.rings.integer', 'Integer')
<class 'sage.rings.integer.Integer'>

Now we horribly break the pickling system:

sage: register_unpickle_override('sage.rings.integer', 'Integer', Rational, call_˓→name=('sage.rings.rational', 'Rational'))
sage: unpickle_global('sage.rings.integer', 'Integer')
<class 'sage.rings.rational.Rational'>

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:

```
sage: unpickle_global('sage.all', 'some_old_class')
Traceback (most recent call last):
... ImportError: cannot import some_old_class from sage.all, call register_unpickle_override('sage.all', 'some_old_class', ...) to fix this
```

```
sage: unpickle_global('sage.some_old_module', 'some_old_class')
Traceback (most recent call last):
... ImportError: cannot import some_old_class from sage.some_old_module, call register_unpickle_override('sage.some_old_module', 'some_old_class', ...) to fix this
```

### 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.

```
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:
a string

EXAMPLES:

```python
sage: def f(N): return N+1
...
sage: g = pickle_function(f)
sage: h = unpickle_function(g)
sage: h(10)
11
```

```python
sage.misc.fpickle.reduce_code(co)
```

EXAMPLES:

```python
sage: def foo(N): return N+1
sage: sage.misc.fpickle.reduce_code(foo.__code__)
(<cyfunction code_ctor at ...>, ...)
```

```python
sage.misc.fpickle.unpickleMethod(im_name, __self__, im_class)
```
support function for copyreg to unpickle method refs

```python
sage.misc.fpickle.unpickleModule(name)
```
support function for copyreg to unpickle module refs

```python
sage.misc.fpickle.unpickle_function(pickled)
```
Unpickle a pickled function.

EXAMPLES:

```python
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))
pg_make_integer = unpickle_global('sage.rings.integer', 'make_integer')
pg_make_integer('c1p')
sage: explain_pickle(dumps(polygen(QQ)))
pg_Polynomial_rational_flint = unpickle_global('sage.rings.polynomial.polynomial_rational_flint', 'Polynomial_rational_flint')
pg_unpickle_PolynomialRing = unpickle_global('sage.rings.polynomial.polynomial_ring_constructor', 'unpickle_PolynomialRing')
pg_RationalField = unpickle_global('sage.rings.rational_field', 'RationalField')
```

(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('clip')

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`.

However, this class lookup process can be customized (using `sage.misc.persist.register_unpickle_override`). For instance, if some future version of Sage renamed `sage/rings/integer.pyx` to `sage/rings/knuth_was_here.pyx`, old pickles would no longer work unless `register_unpickle_override` was used; in that case, GLOBAL of `sage.rings.integer`, `integer` would mean `sage.rings.knuth_was_here.integer`.

By default, `explain_pickle` will map this GLOBAL instruction to `unpickle_global('sage.rings.integer', 'integer')`. Then when this code is evaluated, `unpickle_global` will look up the current mapping in the `register_unpickle_override` table, so the generated code will continue to work even in hypothetical future versions of Sage where `integer.pyx` has been renamed.
If you pass the flag `in_current_sage=True`, then `explain_pickle` will generate code that may only work in the current version of Sage, not in future versions. In this case, it would generate:

```python
from sage.rings.integer import integer
```

and if you run `explain_pickle` in hypothetical future sage, it would generate:

```python
from sage.rings.knuth_was_here import integer
```

but the current code wouldn’t work in the future sage.

If you pass the flag `default_assumptions=True`, then `explain_pickle` will generate code that would work in the absence of any special unpickling information. That is, in either current Sage or hypothetical future Sage, it would generate:

```python
from sage.rings.integer import integer
```

The intention is that `default_assumptions` output is prettier (more human-readable), but may not actually work; so it is only intended for human reading.

There are several functions used in the output of `explain_pickle`. Here I give a brief description of what they usually do, as well as how to modify their operation (for instance, if you’re trying to get old pickles to work).

- **unpickle_global(module, classname)**: unpickle_global('sage.foo.bar', 'baz') is usually equivalent to `sage.foo.bar.baz`, but this can be customized with register_unpickle_override.

- **unpickle_newobj(klass, args)**: Usually equivalent to `klass.__new__(klass, *args)`. If `klass` is a Python class, then you can define `__new__()` to control the result (this result actually need not be an instance of `klass`). (This doesn’t work for Cython classes.)

- **unpickle_build(obj, state)**: If `obj` has a `__setstate__()` method, then this is equivalent to `obj.__setstate__(state)`. Otherwise uses state to set the attributes of `obj`. Customize by defining `__setstate__()`.

- **unpickle_instantiate(klass, args)**: Usually equivalent to `klass(*args)`. Cannot be customized.

- **unpickle_appends(lst, vals)**: Appends the values in `vals` to `lst`. If not `isinstance(lst, list)`, can be customized by defining a `append()` method.

```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_BINPUT(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:

```sage
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:

```sage
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:
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))
{sageleton:7}
sage: pe.stack[-1]._sie_share
True

push_mark()
Push a ‘mark’ onto the virtual machine’s stack.

EXAMPLES:

sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=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:

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:

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
def from_sage.misc.explain_pickle import *
sage: v = TestAppendList()
sage: v.append(7)
Traceback (most recent call last):
... TypeError: ...append() takes 1 positional argument but 2 were given
```

We can still append by directly using the list method:

```sage
def list.append(v, 7)
sage: v
[7]
```

extend()
A deliberately broken extend method.

EXAMPLES:

```sage
def from_sage.misc.explain_pickle import *
sage: v = TestAppendList()
sage: v.extend([3,1,4,1,5,9])
Traceback (most recent call last):
... TypeError: ...extend() takes 1 positional argument but 2 were given
```
We can still extend by directly using the list method:

```python
sage: list.extend(v, (3,1,4,1,5,9))
sage: v
[3, 1, 4, 1, 5, 9]
```

class sage.misc.explain_pickle.TestAppendNonlist
Bases: object

A list-like class, carefully designed to test exact unpickling behavior. Used for testing explain_pickle.

class sage.misc.explain_pickle.TestBuild
Bases: object

A simple class with a __getstate__ but no __setstate__. Used for testing explain_pickle.

class sage.misc.explain_pickle.TestBuildSetstate
Bases: sage.misc.explain_pickle.TestBuild

A simple class with a __getstate__ and a __setstate__. Used for testing explain_pickle.

class sage.misc.explain_pickle.TestGlobalFunnyName
Bases: object

A featureless new-style class which has a name that's not a legal Python identifier.

EXAMPLES:

```python
sage: from sage.misc.explain_pickle import *
sage: globals()['funny$name'] = TestGlobalFunnyName # see comment at end of file
sage: TestGlobalFunnyName.__name__
'funny$name'
sage: globals()['funny$name'] is TestGlobalFunnyName
True
```

class sage.misc.explain_pickle.TestGlobalNewName
Bases: object

A featureless new-style class. When you try to unpickle an instance of TestGlobalOldName, it is redirected to create an instance of this class instead. Used for testing explain_pickle.

EXAMPLES:

```python
sage: from sage.misc.explain_pickle import *
sage: loads(dumps(TestGlobalOldName()))
TestGlobalNewName
```

class sage.misc.explain_pickle.TestGlobalOldName
Bases: object

A featureless new-style class. When you try to unpickle an instance of this class, it is redirected to create a TestGlobalNewName instead. Used for testing explain_pickle.

EXAMPLES:

```python
sage: from sage.misc.explain_pickle import *
sage: loads(dumps(TestGlobalOldName()))
TestGlobalNewName
```

class sage.misc.explain_pickle.TestReduceGetinitargs
Bases: object

5.4. A tool for inspecting Python pickles
An old-style class with a \_\_getinitargs\_ method. Used for testing explain\_pickle.

```python
class sage.misc.explain_pickle.TestReduceNoGetinitargs
```

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 preparser; if False, then produce standard Python code; if None, then produce code that will work either with or without the preparser. (default: True)
- `pedantic` – if True, then carefully ensures that the result has at least as much sharing as the result of cPickle (it may have more, for immutable objects). (default: False)

Exactly one of `pickle` (a string containing a pickle) or `file` (the filename of a pickle) must be provided.

**EXAMPLES:**

```python
sage: explain_pickle(dumps({('a', 'b'): [1r, 2r]}))
{'a': 1r, 'b': 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')
sage: explain_pickle(dumps((s, s)), pedantic=True)
si = 'hi'
(s, 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')
```

(continues on next page)
The `make_rational` function from the `sage.rings.rational` module is used to convert a string representation of a rational number into an object of the `sage.rings.rational.Rational` class. Here are some examples:

```python
from sage.rings.rational import 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')
```

The `explain_pickle_string` function in the `sage.misc.explain_pickle` module is a helper function for `explain_pickle`. It takes a decompressed pickle string as input, and its options are similar to those of `explain_pickle`.

**EXAMPLES:**

```python
sage: sage.misc.explain_pickle.explain_pickle_string(dumps("Hello, world",
˓→compress=False))
'Hello, world'
```

This function is useful for testing whether a string is a valid Python identifier. It uses a conservative test that only allows ASCII identifiers.

**EXAMPLES:**

```python
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
```

The `test_pickle` function in the `sage.misc.explain_pickle` module tests `explain_pickle` on a given pickle. It 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

**EXAMPLES:**

```python
sage: from sage.misc.explain_pickle import test_pickle
sage: test_pickle(['a'])
# py2
0: \x80 PROTO 2
```

5.4. A tool for inspecting Python pickles
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
sage: v = []
sage: unpickle_appends(v, (1, 2, 3))
sage: v
[1, 2, 3]
```

sage.misc.explain_pickle.unpickle_build(obj, state)

Set the state of an object. Used by explain_pickle.

EXAMPLES:

```python
sage: from sage.misc.explain_pickle import *
sage: v = EmptyNewstyleClass()
sage: unpickle_build(v, {'hello': 42})
sage: v.hello
42
```

sage.misc.explain_pickle.unpickle_extension(code)

Takes an integer index and returns the extension object with that index. Used by explain_pickle.

EXAMPLES:

```python
sage: from copyreg import *
sage: add_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 42)
sage: unpickle_extension(42)
<class 'sage.misc.explain_pickle.EmptyNewstyleClass'>
sage: remove_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 42)
```

sage.misc.explain_pickle.unpickle_instantiate(fn, args)

Instantiate a new object of class fn with arguments args. Almost always equivalent to fn(*args). Used by explain_pickle.

EXAMPLES:

```python
sage: unpickle_instantiate(Integer, ('42',))
42
```

sage.misc.explain_pickle.unpickle_newobj(klass, args)

Create a new object; this corresponds to the C code klass->tp_new(klass, args, NULL). Used by explain_pickle.
EXAMPLES:

```
sage: unpickle_newobj(tuple, ([1, 2, 3],))
(1, 2, 3)
```

`sage.misc.explain_pickle.unpicklepersistent(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:

```
sage: from sage.misc.nested_class import A1, nested_pickle
sage: A1.A2.A3.__name__
'A3'
```
All of this is not perfect. In the following scenario:

```
sage: class A1:
    ....:    class A2:
    ....:        pass
sage: class B1:
    ....:    A2 = A1.A2

sage: nested_pickle(A1)
<class '__main__.A1'>
sage: nested_pickle(B1)
<class '__main__.B1'>
sage: A1.A2
<class '__main__.A1.A2'>
sage: B1.A2
<class '__main__.A1.A2'>
```

The name for "A1.A2" could potentially be set to "B1.A2". But that will work anyway.

```
sage: class A1:
    ....:    class A2:
    ....:        pass
sage: class B1:
    ....:    A2 = A1.A2

sage: nested_pickle(A1)
<class '__main__.A1'>
sage: nested_pickle(B1)
<class '__main__.B1'>
sage: A1.A2
<class '__main__.A1.A2'>
sage: B1.A2
<class '__main__.A1.A2'>
```

The name for "A1.A2" could potentially be set to "B1.A2". But that will work anyway.

```
sage: sage.misc.nested_class.modify_for_nested_pickle(cls, name_prefix, module, first_run=True)
    Modify the subclasses of the given class to be picklable, by giving them a mangled name and putting the mangled name in the module namespace.

INPUT:
    • cls – the class to modify
    • name_prefix – the prefix to prepend to the class name
    • module – the module object to modify with the mangled name
    • first_run – optional bool (default True): whether or not this function is run for the first time on cls

NOTE:
    This function would usually not be directly called. It is internally used in NestedClassMetaclass.
```

EXAMPLES:
Here we demonstrate the effect of the `first_run` argument:

```python
sage: modify_for_nested_pickle(A, 'X', module, first_run=False)
sage: A.B.__name__  # nothing changed
'A.B'
sage: A.B.__name__
'A.B'
sage: getattr(module, 'A.B', 'Not found')
<class '__main__.A.B'>
sage: getattr(module, 'X.A.B', 'Not found')
<class '__main__.A.B'>
```

Note that the class is now found in the module under both its old and its new name:

```python
sage: getattr(module, 'X.A.B', 'Not found')
<class '__main__.A.B'>
sage: getattr(module, 'A.B', 'Not found')
<class '__main__.A.B'>
```

`sage.misc.nested_class.nested_pickle(cls)`
This decorator takes a class that potentially contains nested classes. For each such nested class, its name is modified to a new illegal identifier, and that name is set in the module. For example, if you have:

```python
sage: from sage.misc.nested_class import nested_pickle
sage: module = sys.modules['__main__']
sage: class A(object):
    ....:    class B(object):
    ....:        pass
sage: nested_pickle(A)
<class '__main__.A'>
```

then the name of class "B" will be modified to "A.B", and the "A.B" attribute of the module will be set to class "B."

```python
sage: A.B.__name__
'A.B'
sage: getattr(module, 'A.B', 'Not found')
<class '__main__.A.B'>
```

In Python 2.6, decorators work with classes; then `@nested_pickle` should work as a decorator:

```python
sage: from sage.misc.nested_class import *
sage: class A(object):
    ....:    class B(object):
    ....:        pass
```

5.5. Fixing pickle for nested classes 363
sage: @nested_pickle  # todo: not implemented
....: class A2(object):
....:     class B:
....:         pass
sage: A2.B.__name__  # todo: not implemented
'A2.B'
sage: getattr(module, 'A2.B', 'Not found')  # todo: not implemented
<class __main__.A2.B at ...>

EXAMPLES:

sage: from sage.misc.nested_class import *
sage: loads(dumps(MainClass.NestedClass()))  # indirect doctest
<sage.misc.nested_class.MainClass.NestedClass object at 0x...>

class sage.misc.nested_class.NestedClassMetaclas
Bases: type

A metaclass for nested pickling.

Check that one can use a metaclass to ensure nested_pickle is called on any derived subclass:

sage: from sage.misc.nested_class import NestedClassMetaclas
sage: class ASuperClass(object, metaclass=NestedClassMetaclas):
    ....: pass
sage: class A3(ASuperClass):
    ....: class B(object):
    ....:     pass
sage: A3.B.__name__
'A3.B'
sage: getattr(sys.modules['__main__'], 'A3.B', 'Not found')
<class '__main__.A3.B'>

class sage.misc.nested_class.MainClass
Bases: object

A simple class to test nested_pickle.

EXAMPLES:

sage: from sage.misc.nested_class import *
sage: loads(dumps(MainClass()))
<sage.misc.nested_class.MainClass object at 0x...>

class NestedClass
Bases: object

EXAMPLES:

sage: from sage.misc.nested_class import *
sage: loads(dumps(MainClass.NestedClass()))
<sage.misc.nested_class.MainClass.NestedClass object at 0x...>

class NestedSubClass
Bases: object

EXAMPLES:
```python
sage: from sage.misc.nested_class import *
sage: loads(dumps(MainClass.NestedClass.NestedSubClass()))
<sage.misc.nested_class.MainClass.NestedClass.NestedSubClass object at 0x...>
sage: getattr(sage.misc.nested_class, 'MainClass.NestedClass.NestedSubClass')
<class 'sage.misc.nested_class.MainClass.NestedClass.NestedSubClass'>
sage: MainClass.NestedClass.NestedSubClass.__name__
'MainClass.NestedClass.NestedSubClass'
```

dummy(x, r=(1, 2, 3.4), *args, **kwds)

A dummy method to demonstrate the embedding of method signature for nested classes.

## 5.6 Loading and saving sessions and listing all variables

### EXAMPLES:

We reset the current session, then define a rational number 2/3, and verify that it is listed as a newly defined variable:

```python
sage: reset()
sage: w = 2/3; w
2/3
sage: show_identifiers()
['w']
```

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!

```python
sage: save_session(os.path.join(SAGE_TMP, 'session'))
```

This saves a dictionary with `w` as one of the keys:

```python
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:

```python
sage: reset()
sage: show_identifiers()
[]
sage: load_session(os.path.join(SAGE_TMP, 'session'))
```

Indeed `w` is now defined again:

```python
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:**

```python
sage: reset()
sage: w = 10
sage: show_identifiers()
['w']
```

When we call `init()` below it reinitializes the internal table, so the `w` we just defined doesn’t count as a new identifier:

```python
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:**

```python
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.

```python
sage: tmp_f = tmp_filename()
sage: save_session(tmp_f)
sage: del a; del f
sage: load_session(tmp_f)
sage: print(a)
5
```

Note that `f` does not come back, since it is a function, hence couldn’t be saved:

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

   ```python
   class Foo: pass
   ```

   and make an instance with:

   ```python
   f = Foo()
   ```

   Then save_session() followed by quit and load_session() fails. I doubt there is any good way to deal with this. Fortunately, one can simply re-evaluate the code to define Foo, and suddenly load_session() works fine.

**INPUT:**

- **name** – string (default: ‘sage_session’) name of sobj to save the session to.
- **verbose** – bool (default: False) if True, print info about why certain variables can’t be saved.

**OUTPUT:**

- Creates a file and returns silently.

**EXAMPLES:**

For testing, we use a temporary file that will be removed as soon as Sage is left. Of course, for permanently saving your session, you should choose a permanent file.

```python
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:

```python
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:
```python
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 ...
```

```python
sage.misc.session.show_identifiers(hidden=False)
```

Returns a list of all variable names that have been defined during this session. By default, this returns only those identifiers that don’t start with an underscore.

**INPUT:**

- hidden – bool (Default: False); If True, also return identifiers that start with an underscore.

**OUTPUT:**

A list of variable names

**EXAMPLES:**

We reset the state of all variables, and see that none are defined:

```python
sage: reset()
sage: show_identifiers()
[]
```

We then define two variables, one which overwrites the default factor function; both are shown by `show_identifiers()`:

```python
sage: a = 10
sage: factor = 20
sage: show_identifiers()
['a', 'factor']
```

To get the actual value of a variable from the list, use the `globals()` function:

```python
sage: globals()['factor']
20
```

By default `show_identifiers()` only returns variables that don’t start with an underscore. There is an option hidden that allows one to list those as well:

```python
sage: _hello = 10
sage: show_identifiers()
['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:

```python
sage: show_identifiers(hidden=True)  # random output
['__', '_i', '_6', '_4', '_3', '_1', '_ii', '_doc__', '_builtins__', '__', '_9', ...
'doc__', '_i2', '_i4', 'factor', '__file__', '_hello', '_i13', '_i11', '_i10', '_i5', '_i3', '_13', '_10', '_iii', '_i9', '_i8', '_i7', '_i6', '_
_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:

```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:

```sage
print(sage.misc.banner.banner_text(full=True))
+——————————————————————–+
| SageMath version ... |
print(sage.misc.banner.banner_text(full=False))
SageMath version ..., Release Date: ...
```
sage.misc.banner.require_version(major, minor=0, tiny=0, prerelease=False, print_message=False)

True if Sage version is at least major.minor.tiny.

INPUT:

- major – integer
- minor – integer (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:

```python
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:

```python
sage: version()
'SageMath version ..., Release Date: ...'
```

sage.misc.banner.version_dict()

A dictionary describing the version of Sage.

INPUT:

nothing

OUTPUT:


This process the Sage version string and produces a dictionary. It expects the Sage version to be in one of these forms:
where ‘N’ stands for an integer and ‘str’ stands for a string. The first integer is stored under the ‘major’ key and
the second integer under ‘minor’. If there is one more integer, it is stored under ‘tiny’; if there are two more
integers, then they are stored together as a float N.N under ‘tiny’. If there is a string, then the key ‘prerelease’
returns True.

For example, if the Sage version is ‘3.2.1’, then the dictionary is {‘major’: 3, ‘minor’: 2, ‘tiny’: 1, ‘prerelease’:
False}. If the Sage version is ‘3.2.1.2’, then the dictionary is {‘major’: 3, ‘minor’: 2, ‘tiny’: 1.200..., ‘prerelease’:
False}. If the Sage version is ‘3.2.alpha0’, then the dictionary is {‘major’: 3, ‘minor’: 2, ‘tiny’: 0, ‘prerelease’:
True}.

EXAMPLES:

```python
sage: from sage.misc.banner import version_dict
sage: print("SageMath major version is %s% version_dict()['major']")
SageMath major version is ...
sage: version_dict()['major'] == int(sage.version.version.split('.0') [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:**

```python
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
```

(continues on next page)
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

```python
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
```

```python
def 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()
sage: viewer.browser('open -a /Applications/Firefox.app')  # indirect doctest
sage: viewer.browser()  # open -a /Applications/Firefox.app
sage: viewer.browser(old_browser)  # restore old value
```

```python
def 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
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 arg 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 arg 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.dviViewer()**

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.pdfViewer()**

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.pngViewer()**

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.

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

sage.misc.sagedoc.detex(s, embedded=False)

This strips LaTeX commands from a string; it is used by the format function to process docstrings for display from the command line interface.

INPUT:

- s - string
- embedded - boolean (optional, default False)

If embedded is False, then do the replacements in both math_substitutes and nonmath_substitutes. If True, then only do nonmath_substitutes.

OUTPUT:

string

EXAMPLES:

sage: from sage.misc.sagedoc import detex
sage: detex(r'Some math: \text{n} \geq k'. A website: \url{sagemath.org}.)
'Some math: n >= k. A website: sagemath.org.'

sage: detex(r'More math: \text{x} \mapsto y'. \text{Bold face}.)
'More math: x \rightarrow y. \text{Bold face}.'
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>>> or, 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>>>\n'identity_matrix>>>\n'sage: format("<<<identity_matrix>>>\n'...Definition: identity_matrix(...'
```

6.1. Interactive Sage Sessions
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:

```
sage: from sage.misc.sagedoc import format_search_as_html
sage: format_search_as_html('Source', 'algebras/steenrod_algebra_element.py:
  an antihomomorphism: if we call the antipode 'c', then', 'antipode
  antihomomorphism')
  
  \html\text{Search Source: "antipode antihomomorphism"/h2/>\text{Search Other: "antipode antihomomorphism"/h2/>}

sage: format_search_as_html('Other', 'html/en/reference/sage/algebras/steenrod_algebra_element.html:an antihomomorphism: if we call the antipode <span class="math">c</span>, then', 'antipode antihomomorphism')

\html\text{Search Other: "antipode antihomomorphism"/h2/>\text{Search Other: "antipode antihomomorphism"/h2/>}
```

sage.misc.sagedoc.format_src(s)

Format Sage source code s for viewing with IPython.

If s contains a string of the form "<<<obj>>>", then it replaces it with the source code for "obj".

INPUT: s - string

OUTPUT: string

EXAMPLES:

```
sage: from sage.misc.sagedoc import format_src
sage: format_src('unladen swallow')
'unladen swallow'

sage: format_src('<<<Sq>>>')[5:15]
'Sq(*nums):'
```
sage.misc.sagedoc.help(module=None)
If there is an argument module, print the Python help message for module. With no argument, print a help message about getting help in Sage.

EXAMPLES:

```sage
sage: help()
Welcome to Sage ...
```

sage.misc.sagedoc.my_getsource(obj, oname=None)
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
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
sage: from sage.misc.sagedoc import process_dollars
sage: process_dollars('hello')
'hello'
sage: process_dollars('some math: $x=y$')
'some math: `x=y`'
sage: process_dollars(r'a `\$REAL\` dollar sign: \$')
'a `\$REAL\` dollar sign: $'
sage: s = '\n first line\n indented $x=y$'
sage: s == process_dollars(s)
True
```

Don’t replace dollar signs enclosed in curly braces:
This is not perfect:

```sage
sage: process_dollars(r'f(n) = 0 \text{ if } n\text{ is prime}')
'f(n) = 0 \text{ if } n\text{ 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.miss.sagedoc.process_extlinks(s, embedded=False)
In docstrings at the command line, process markup related to the Sphinx extlinks extension. For example, replace :trac:`1234` with https://trac.sagemath.org/1234, 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 :func:`format()`, and if in the notebook, it sets embedded to be True, otherwise False.

**EXAMPLES:**

```sage
sage: from sage.misc.sagedoc import process_extlinks
sage: process_extlinks(r'\text{See :trac:`1234`, :wikipedia:`Wiki...'}')

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...'}')
'see https://docs.python.org/release/.../reference/datamodel.html#implementing...
```sage.miss.sagedoc.process_mathtt(s)
Replace `\texttt{BLAH}` with BLAH in the command line.

**INPUT:**

- s - string, in practice a docstring

This function is called by :func:`format()`.

**EXAMPLES:**

```sage
sage: from sage.misc.sagedoc import process_mathtt
sage: process_mathtt(r'e^\mathtt{self}')
e^self
```

sage.miss.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 *search_src()* for more examples.

```python
sage: print(search_def("fetch", interact=False)) # random # long time
matrix/matrix0.pyx: cdef fetch(self, key):

sage: print(search_def("fetch", path_re="pyx", interact=False)) # random # long time
matrix/matrix0.pxd: cdef fetch(self, key)
```

`sage.misc.sagedoc.search_doc(string, extra1="", extra2="", extra3="", extra4="", extra5="", **kwds)`

Search Sage HTML documentation for lines containing `string`. The search is case-insensitive by default.

The file paths in the output are relative to `$SAGE_DOC`.

INPUT: same as for *search_src()*.

OUTPUT: same as for *search_src()*.

**EXAMPLES:**
See the documentation for *search_src()* for more examples.

```python
sage: search_doc('creates a polynomial', path_re='tutorial', interact=False) # optional - sagemath_doc_html, long time
html/en/tutorial/tour_polynomial.html:<p>This creates a polynomial ring and tells Sage to use (the string)
```

If you search the documentation for ‘tree’, then you will get too many results, because many lines in the documentation contain the word ‘tocctree’. If you use the whole_word option, though, you can search for ‘tree’ without returning all of the instances of ‘tocctree’. In the following, since `search_doc('tree', interact=False)` returns a string with one line for each match, counting the length of `search_doc('tree', interact=False).splitlines()` gives the number of matches.

```python
sage: N = len(search_doc('tree', interact=False).splitlines()) # optional - sagemath_doc_html, long time

sage: L = search_doc('tree', whole_word=True, interact=False).splitlines() # optional - sagemath_doc_html, long time

sage: len(L) < N # optional - sagemath_doc_html, 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 - sagemath_doc_html, long time
True
```

`sage.misc.sagedoc.search_src(string, extra1="", extra2="", extra3="", extra4="", extra5="", **kwds)`

Search Sage library source code for lines containing `string`. The search is case-insensitive by default.

INPUT:
• string - a string to find in the Sage source code.
• extral,...,extra5 - additional strings to require when searching. Lines must match all of these, as well as string.
• whole_word (optional, default False) - if True, search for string and extral (etc.) as whole words only. This assumes that each of these arguments is a single word, not a regular expression, and it might have unexpected results if used with regular expressions.
• ignore_case (optional, default 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))
Traceback (most recent call last):
...
error: missing ), unterminated subpattern at position 6
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):
matrix/matrix0.pxd:
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
˓→# long time
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))
misc/sagedoc.py:... print(search_src(r" fetch\(", "def", interact=False)) # random
˓→# long time
misc/sagedoc.py:... print(search_src(r" fetch\(", "def", "pyx", interact=False)) #␣
˓→random # long time

(continues on next page)

6.1. Interactive Sage Sessions

383


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\n\n\n\nsage: test2 = ' TESTS:: \n
sage: refs = ' REFERENCES: \n\n```

Test lines starting with “REFERENCES:”:

```python
sage: skip_TESTS_block(start + test + refs).rstrip() == (start + refs).rstrip()
True
sage: skip_TESTS_block(start + test + test2 + refs).rstrip() == (start + refs).rstrip()
True
sage: skip_TESTS_block(start + test + refs + test2).rstrip() == (start + refs).rstrip()
True
```
Test Sphinx directives:

```python
sage: directive = ' .. todo::
        do some stuff
'
sage: skip_TESTS_block(start + test + refs + test2 + directive).rstrip() == (start
        + refs + directive).rstrip()
True
```

Test unindented lines:

```python
sage: unindented = 'NOT INDENTED
'
sage: skip_TESTS_block(start + test + unindented).rstrip() == (start + unindented).
        rstrip()
True
sage: skip_TESTS_block(start + test + unindented + test2 + unindented).rstrip() ==
        (start + unindented + unindented).rstrip()
True
```

Test headers:

```python
sage: header = ' Header:
~~~~~~~~
'
sage: 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
sage: fake_header = ' Header:
        -=-==-=-=
'
sage: 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
sage: another_fake = ' blah
        ----
'
sage: skip_TESTS_block(start + test + another_fake).rstrip() == start.rstrip()
True
```

Double colons :: are also not considered as headers (trac ticket #27896):

```python
sage: colons = '::

sage: 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

```
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:

```
sage: from sage.misc.sphinxify import sphinxify
sage: sphinxify('A test')
'<div class="docstring">

<p>A test</p>

</div>'
sage: sphinxify('**Testing**

`monospace`
')
'<div class="docstring">

<p>Testing</p>

</div>'
sage: sphinxify('`x=y`')
'<div class="docstring">

<p>x=y</p>

</div>'
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:

```
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

class sage.misc.package.PackageInfo
    Bases: tuple
    Represents information about a package.

    installed_version
        Alias for field number 3

    is_installed()
        Whether the package is installed in the system.

    name
        Alias for field number 0

    remote_version
        Alias for field number 4

    source
        Alias for field number 2

    type
        Alias for field number 1

exception 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:

    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:**

```python
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:**

```python
sage: sorted(installed_packages().keys())  # optional - build
[...'gmpy2', ...'sage_conf', ...]
sage: installed_packages()['gmpy2']  # optional - build, random
'2.1.0b5'
```

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:**

```python
sage: is_package_installed('gap')  # optional - build
True
```

Giving just the beginning of the package name is not good enough:

```python
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_sources, local, ignore_URLError, exclude_pip, *pkg_types)`

Return a dictionary of information about each package.
The keys are package names and values are named tuples with the following keys:

- 'type': either 'base', 'standard', 'optional', or 'experimental'
- 'source': either 'normal', `pip`, or 'script'
- 'installed': boolean
- 'installed_version': None or a string
- '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
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.is_installed()  # optional - build
True

sage: sage_conf_info.source  # optional - build
'script'

sage: L = list_packages(pkg_sources=['pip'], local=True)  # optional - build

sage: bs4_info = L['beautifulsoup4']  # optional - build internet
sage: bs4_info.type  # optional - build internet
'optional'

sage: bs4_info.source  # optional - build internet
'pip'

sage: [p for p, d in list_packages('optional', exclude_pip=True).items() if d.source == 'pip']
[]
```

Check the option `exclude_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:

```
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:

```
sage: from sage.misc.package import package_manifest
sage: sagetex_manifest = package_manifest('sagetex')  # optional - build
sage: sagetex_manifest['package_name'] == 'sagetex'  # optional - build
True
sage: 'files' in sagetex_manifest  # optional - build
True
```

Test a nonexistent package:

```
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:

```python
def std = package_versions(['standard', local=True]) # optional - build
def 'gap' in std # optional - build
True
def std['zn_poly'] # optional - build, random
('0.9.p12', '0.9.p12')
```

sage.misc.package.pip_installed_packages(normalization=None)

Return a dictionary name -> version of installed pip packages.

This command returns all pip-installed packages. Not only Sage packages.

INPUT:

- normalization – (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:

```python
def from sage.misc.package import pip_installed_packages
def d = pip_installed_packages() # optional - build
def 'scipy' in d # optional - build
True
def d['scipy'] # optional - build
'...'
def d['beautifulsoup4'] # optional - build beautifulsoup4
'...'
def d['prompt-toolkit'] # optional - build
'...'
def d = pip_installed_packages(normalization='spkg') # optional - build
'
```
These tests are reliable since the tested package does not exist:

```python
sage: nap = 'hey_this_is_NOT_a_python_package'
sage: pypi = 'http://this.is.not.pypi.com/'
sage: pip_remote_version(nap, pypi_url=pypi, ignore_URLError=True) # optional - internet
doctest:...: UserWarning: failed to fetch the version of pkg='hey_this_is_NOT_a_python_package' at http://this.is.not.pypi.com/.../json
```

```python
sage: pip_remote_version(nap, pypi_url=pypi, ignore_URLError=False) # optional - internet
Traceback (most recent call last):
... HTTPError: HTTP Error 404: Not Found
```

```
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
g sage: pkgname_split('hello_world-1.2')
['hello_world', '1.2']
```
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:**
6.3 Credits

6.3.1 Dependency usage tracking for citations

sage.misc.citation.cython_profile_enabled()
    Return whether Cython profiling is enabled.

    EXAMPLES:

    sage: from sage.misc.citation import cython_profile_enabled
    sage: cython_profile_enabled() # random
    False
    sage: type(cython_profile_enabled()) is bool
    True

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:

    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

```python
class sage.misc.copying.License
    Bases: object
```

6.3. Credits

395
CHAPTER SEVEN

DEVELOPMENT TOOLS

7.1 Testing

7.1.1 Unit testing for Sage objects

```python
class sage.misc.sage_unittest.InstanceTester(instance=None, 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")
    hello world
```

397
runTest()
Trivial implementation of unittest.TestCase.runTest() to please the super class TestCase. That’s the price to pay for abusively inheriting from it.

EXAMPLES:

```python
sage: from sage.misc.sage_unittest import InstanceTester
sage: tester = InstanceTester(ZZ, verbose = True)
sage: tester.runTest()
```

some_elements(S=None, repeat=None)
Return a list (or iterable) of elements of the instance on which the tests should be run.
This is only meaningful for container objects like parents.

INPUT:

- S – a set of elements to select from. By default this will use the elements passed to this tester at creation time, or the result of some_elements() if no elements were specified.
- repeat – integer (default: None). If given, instead returns a list of tuples of length repeat from S.

OUTPUT:

A list of at most self._max_runs elements of $S^r$, or a sample of at most self._max_samples if that is not None.

EXAMPLES:

By default, this calls some_elements() on the instance:

```python
sage: from sage.misc.sage_unittest import InstanceTester
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)
```
The `repeat` keyword can give pairs or triples from `S`:

```python
sage: list(tester.some_elements(repeat=2))
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1)]
```

You can use `max_samples` to sample at random, instead of in order:

```python
sage: 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:

```python
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)
```

(continues on next page)
```python
sage: len(C)
390625
sage: tester = InstanceTester(C, elements = C, max_runs=4)
sage: list(tester.some_elements())
[(0, 0, 0, 0), (0, 0, 0, 1), (0, 0, 0, 2), (0, 0, 0, 3)]
```

```python
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:

```python
sage: TestSuite("bla").run()
```

```python
class sage.misc.sage_unittest.TestSuite(instance)
Bases: object
Test suites for Sage objects.
EXAMPLES:

```python
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:

```python
sage: TestSuite(1).run(allow_time_limit=False, verbose = True)
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_nonzero_equal() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
```

Those methods are typically implemented by abstract super classes, in particular via categories, in order to enforce standard behavior and API, or provide mathematical sanity checks. For example if self is in the category of finite semigroups, this checks that the multiplication is associative (at least on some elements):

```python
sage: S = FiniteSemigroups().example(alphabet = ('a', 'b'))
sage: TestSuite(S).run(allow_time_limit=False, verbose = 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 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
```

(continues on next page)
The different test methods can be called independently:

`sage: S._test_associativity()`

Debugging tip: in case of failure of some test, use `%pdb on` to turn on automatic debugging on error. Run the failing test 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:

`sage: S._test_associativity(elements = S)`

Adding a new test boils down to adding a new method in the class of the object or any super class (e.g. in a category). This method should use the utility `_tester()` to handle standard options and report test failures. See the code of `_test_an_element()` for an example. Note: Python’s testunit convention is to look for methods called `test*`; we use instead `_test_*` so as not to pollute the object’s interface.

Eventually, every implementation of a `SageObject` should run a `TestSuite` on one of its instances in its doctest (replacing the current `loads(dumps(x))` tests).

Finally, running `TestSuite` on a standard Python object does some basic sanity checks:

`sage: TestSuite(int(1)).run(\texttt{verbose = True})`

```python
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(...)\texttt{.run()}` is called under the debugger, or with `%pdb on` (how to detect this? see `get_ipython()`). 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

7.1. Testing
Add some standard option \texttt{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 \texttt{._test_commutative} method of the category \texttt{CommutativeSemigroups}() may just check that the provided generators commute, implicitly assuming that generators indeed generate the semigroup (as required by \texttt{Semigroups}()).

\texttt{run(category=None, skip=[], catch=True, raise_on_failure=False, **options)}

Run all the tests from this test suite:

INPUT:

\begin{itemize}
  \item \texttt{category} - a category; reserved for future use
  \item \texttt{skip} - a string or list (or iterable) of strings
  \item \texttt{raise_on_failure} - a boolean (default: False)
  \item \texttt{catch} - a boolean (default: True)
\end{itemize}

All other options are passed down to the individual tests.

EXAMPLES:

\begin{verbatim}
sage: TestSuite(ZZ).run()
\end{verbatim}

We now use the \texttt{verbose} option:

\begin{verbatim}
sage: TestSuite(1).run(\texttt{verbose = True})
\end{verbatim}

Some tests may be skipped using the \texttt{skip} option:

\begin{verbatim}
sage: TestSuite(1).run(\texttt{verbose = True, skip ="._test_pickling"})
sage: TestSuite(1).run(\texttt{verbose = True, skip =["._test_pickling", "._test_category" \texttt{\rightarrow}])
\end{verbatim}

We now show (and test) some standard error reports:

\begin{verbatim}
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()
\end{verbatim}

(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 __main__.Blah... failed
------------------------------------------------------------
The following tests failed: _test_b, _test_d, _test_pickling
sage: TestSuite(Blah()).run(verbos=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 __main__.Blah... failed
------------------------------------------------------------
The following tests failed: _test_b, _test_d, _test_pickling
File "/opt/sage/local/lib/python/site-packages/sage/misc/sage_unittest.py",...
-->line 183, in run
test_method(tester = tester)
The catch=False option prevents TestSuite from catching exceptions:
sage: TestSuite(Blah()).run(catch=False)
Traceback (most recent call last):
...
In conjunction with `%pdb` on, this allows for the debugger to jump directly to the first failure location.

```
File ..., in _test_b
    def _test_b(self, tester): tester.fail()
...
AssertionError: None
```

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 \texttt{verbose} (defaulting to false). With \texttt{verbose} true, it should print the values being tested. Suppose \texttt{test\_foo()} takes an argument for number of iterations. Then the doctests could be:

\begin{verbatim}
    test\_foo(2, verbose=True, seed=0)
    test\_foo(10)
    test\_foo(100) # long time
\end{verbatim}

The first doctest, with the specified seed and \texttt{verbose=True}, simply verifies that the tests really are reproducible (that \texttt{test\_foo} is correctly using the \texttt{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):

\begin{verbatim}
    for _ in xrange(10^10): test\_foo(100)
\end{verbatim}

instead of:

\begin{verbatim}
    test\_foo(10^12)
\end{verbatim}

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 \texttt{test\_foo(100)} with a known-failing random seed.

See \texttt{sage.misc.random\_testing.test\_add\_commutes()} for a simple example using this decorator, and \texttt{sage.rings.tests} for realistic uses.

Setting \texttt{print\_seed} to true is useless in doctests, because the random seed printed will never match the expected doctest result (and using \texttt{# 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:

\begin{verbatim}
    while True: test\_foo(print\_seed=True)
\end{verbatim}

and look at the last seed that was printed before it crashed.

\texttt{sage.misc.random\_testing.test\_add\_commutes(*args, **kwargs)}

This is a simple demonstration of the \texttt{random\_testing()} decorator and its recommended usage.

We test that addition is commutative over rationals.

**EXAMPLES:**

\begin{verbatim}
    sage: from sage.misc.random\_testing import test\_add\_commutes
    sage: test\_add\_commutes(2, verbose=True, seed=0)
    a == -4, b == 0 ...
    Passes!
    a == -1/2, b == -1/95 ...
    Passes!
    sage: test\_add\_commutes(10)
    sage: test\_add\_commutes(1000) # long time
\end{verbatim}
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 $a+b = a\times b$, for $a, b$ rational. This is of course false, so the test will almost always fail.

EXAMPLES:

```
sage: from sage.misc.random_testing import test_add_is_mul
```

We start by testing that we get reproducible results when setting seed to 0.

```
sage: test_add_is_mul(2, verbose=True, seed=0)
a == -4, b == 0 ...
Random testing has revealed a problem in test_add_is_mul
Please report this bug! You may be the first person in the world to have seen this problem.
Please include this random seed in your bug report:
Random seed: 0
AssertionError()
```

Normally in a @random_testing doctest, we would leave off the verbose=True and the # random. We put it in here so that we can verify that we are seeing the exact same error when we reproduce the error below.

```
sage: test_add_is_mul(10, verbose=True) # random
a == -2/7, b == 1 ...
Random testing has revealed a problem in test_add_is_mul
Please report this bug! You may be the first person in the world to have seen this problem.
Please include this random seed in your bug report:
Random seed: 216390410596009428782506007128692114173
AssertionError()
```

OK, now assume that some user has reported a test_add_is_mul() failure. We can specify the same random_seed that was found in the bug report, and we will get the exact same failure so that we can debug the “problem”.

```
sage: test_add_is_mul(10, verbose=True, seed=216390410596009428782506007128692114173)
a == -2/7, b == 1 ...
Random testing has revealed a problem in test_add_is_mul
Please report this bug! You may be the first person in the world to have seen this problem.
Please include this random seed in your bug report:
Random seed: 216390410596009428782506007128692114173
AssertionError()
```
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, ...
...
P = TestParent1()
TestSuite(P).run()
Failure ...
The following tests failed: _test_elements, _test_pickling
```

They actually need to be in the NestedClassMetaClass. However, due to a technical detail, this is currently not directly supported:

```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
```

Instead, the easiest is to inherit from UniqueRepresentation, which is what you want to do anyway most of the time:

```sage
P = TestParent3()
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()
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:
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)^1000001 * (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 \(y^2 = x^3 + 37x - 997\).
```

`sage.misc.benchmark.benchmark(n=-1)`

Run a well-chosen range of Sage commands and record the time it takes for each to run.

**INPUT:**

- `n – int (default: -1)` the benchmark number; the default of -1 runs all the benchmarks.

**OUTPUT:**

- list – summary of timings for each benchmark. int – if n == -1, also return the total time

**EXAMPLES:**

```python
sage: from sage.misc.benchmark import *
sage: _ = benchmark()
Running benchmark 0
Benchmark 0: Factor the following polynomial over the rational numbers: \((x^97+19*x+1)*(x^103-19*x^97+14)*(x^100-1)\)
Time: ... seconds
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}\nTime: ... seconds
Running benchmark 3
Some basic arithmetic with very large Rational numbers: \('(2/3)^{100001} * (17/19)^{-100001}\nTime: ... seconds
Running benchmark 4
Rational polynomial arithmetic using Sage. Compute \((x^29+17*x-5)^200\).
Time: ... seconds
Running benchmark 5
Rational polynomial arithmetic using Sage. Compute \((x^{19} - 18*x + 1)^{50}\) one hundred times.
Time: ... seconds
Running benchmark 6
Compute the p-division polynomials of \(y^2 = x^3 + 37*x - 997\) for primes \(p < 40\).
Time: ... seconds
Running benchmark 7
Compute the Mordell-Weil group of \(y^2 = x^3 + 37*x - 997\).
Time: ... seconds
Running benchmark 8
```
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:

```sage
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

```python
class sage.misc.sage_timeit.SageTimeitResult(stats, series=None)
    Bases: object

    Represent the statistics of a timeit() command.

    Prints as a string so that it can be easily returned to a user.

    INPUT:

    • stats – tuple of length 5 containing the following information:
      – integer, number of loops
      – integer, repeat number
      – Python integer, number of digits to print
      – number, best timing result
      – str, time unit

    EXAMPLES:

    sage: from sage.misc.sage_timeit import SageTimeitResult
    sage: SageTimeitResult( (3, 5, int(8), pi, 'ms') )
    3 loops, best of 5: 3.1415927 ms per loop

    units = [u"s", u"ms", u"\mu s", u"ns"]
    scaling = [1, 1e3, 1e6, 1e9]
    number = 7
    repeat = 13
    precision = int(5)
    best = pi / 10 ** 9
    order = 3
    stats = (number, repeat, precision, best * scaling[order], units[order])
    SageTimeitResult(stats)
    7 loops, best of 13: 3.1416 ns per loop
```

If the third argument is not a Python integer, a `TypeError` is raised:

```sage
sage: SageTimeitResult( (1, 2, 3, 4, 's') )
<repr('<sage.misc.sage_timeit.SageTimeitResult at 0x...>') failed: TypeError: * wants...
  int>
```

```python
sage.misc.sage_timeit.sage_timeit(stmt, globals_dict=None, preparse=None, number=0, repeat=3, precision=3, seconds=False)
```

Accurately measure the wall time required to execute `stmt`.

Chapter 7. Development Tools
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: from sage.misc.sage_timeit import sage_timeit
sage: sage_timeit('3^100000', globals(), preparse=True, number=50)  # random
'50 loops, best of 3: 1.97 ms per loop'
sage: sage_timeit('3^100000', globals(), preparse=False, number=50)  # random
'50 loops, best of 3: 67.1 ns per loop'
sage: a = 10
sage: sage_timeit('a^2', globals(), number=50)  # random
'50 loops, best of 3: 4.26 us per loop'
```
If you only want to see the timing and not have access to additional information, just use the `timeit` object:
```
sage: timeit('10^2', number=50)
50 loops, best of 3: ... per loop
```
Using `sage_timeit` gives you more information though:
```
sage: s = sage_timeit('10^2', globals(), repeat=1000)
sage: len(s.series)
1000
sage: mean(s.series)  # random output
3.1298141479492283e-07
sage: min(s.series)  # random output
2.9258728027343752e-07
sage: t = stats.TimeSeries(s.series)
sage: t.scale(10^6).plot_histogram(bins=20, figsize=[12,6], ymax=2)
```
The input expression can contain newlines (but doctests cannot, so we use `os.linesep` here):
```
sage: from sage.misc.sage_timeit import sage_timeit
sage: from os import linesep as CR
sage: # sage_timeit(r'a = 2\nb=131\nfactor(a^b-1)')
```
(continues on next page)
Test to make sure that `timeit` behaves well with output:

```sage
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
sage: timeit("print('Hi')", seconds=True)  # random output
1.42555236816e-06
sage: t = timeit("print('Hi')", seconds=True)
sage: t  # 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
    sage: timeit('2^10000')
    625 loops, best of 3: ... per loop
    ```

    We illustrate some options:
    ```sage
    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
    sage: timeit('2^10000', preparse=False, number=50)
    50 loops, best of 3: ... per loop
    ```

    The input can contain newlines:
    ```sage
    sage: timeit("a = 2\nb=131\nfactor(a^b-1)", number=25)
    25 loops, best of 3: ... per loop
    ```
See also:

runsnake()

eval(code, globs=None, locals=None, **kwds)

This eval function is called when doing \%timeit in the notebook.

INPUT:

• code – string of code to evaluate; may contain newlines.

• globs – global variables; if not given, uses module scope globals.

• locals – ignored completely.

• kwds – passed onto sage_timeit. Common options are prepare, number, repeat, precision. See sage_timeit() for details.

OUTPUT: string – timing information as a string

EXAMPLES:

sage: timeit.eval("2+2")                  # random output
'625 loops, best of 3: 1.47 us per loop'

We emphasize that timeit times WALL TIME. This is good in the context of Sage where commands often call out to other subprocesses that do not appear in CPU time.

sage: timeit('sleep(float(0.5))', number=3) # long time (5s on sage.math, 2012)
3 loops, best of 3: ... ms per loop

7.2.4 Simple profiling tool

AUTHORS:

• David Harvey (August 2006)

• Martin Albrecht

class sage.misc.profiler.Profiler(systems=[], verbose=False)

Bases: object

Keeps track of CPU time used between a series of user-defined checkpoints.

It’s probably not a good idea to use this class in an inner loop :-)

EXAMPLES:

sage: def f():                        # not tested
....:    p = Profiler()             # not tested

Calling p(message) creates a checkpoint:

sage: p("try factoring 15")        # not tested

Do something time-consuming:

sage: x = factor(15)               # not tested

You can create a checkpoints without a string; Profiler will use the source code instead:
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()   # optional - gperftools
sage: prof.stop()   # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
```
REFERENCE:

Uses the Google performance analysis tools. Note that they are not included in Sage, you have to install them yourself on your system.

AUTHORS:

• Volker Braun (2014-03-31): initial version

class sage.misc.gperftools.Profiler(filename=None)

    Bases: sage.structure.sage_object.SageObject

    Interface to the gperftools profiler

INPUT:

    • filename – string or None (default). The file name to log to. By default, a new temporary file is created.

EXAMPLES:

    sage: from sage.misc.gperftools import Profiler
    sage: Profiler()
    Profiler logging to ...

filename()

    Return the file name

OUTPUT:

    String.

EXAMPLES:

    sage: from sage.misc.gperftools import Profiler
    sage: prof = Profiler()
    sage: prof.filename()
    '../tmp_...perf'

save(filename, cumulative=True, verbose=True)

    Save report to disk.

INPUT:

    • filename – string. The filename to save at. Must end with one of .dot, .ps, .pdf, .svg, .gif, or .txt to specify the output file format.
    • cumulative – boolean (optional, default: True). Whether to return cumulative timings.
    • verbose – boolean (optional, default: True). Whether to print informational messages.

EXAMPLES:

    sage: from sage.misc.gperftools import Profiler, run_100ms
    sage: prof = Profiler()
    sage: prof.start()  # optional - gperftools
    sage: run_100ms()   # optional - gperftools
    sage: prof.stop()   # optional - gperftools
    PROFILE: interrupts/evictions/bytes = ...
    sage: f = tmp_filename(ext='.txt')  # optional - gperftools
    sage: prof.save(f, verbose=False)   # optional - gperftools
start()  
Start profiling  
EXAMPLES:

```
sage: from sage.misc.gperftools import Profiler, run_100ms
sage: prof = Profiler()
sage: prof.start()  # optional - gperftools
sage: run_100ms()
sage: prof.stop()  # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
```

stop()  
Stop the CPU profiler  
EXAMPLES:

```
sage: from sage.misc.gperftools import Profiler, run_100ms
sage: prof = Profiler()
sage: prof.start()  # optional - gperftools
sage: run_100ms()
sage: prof.stop()  # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
```

top(cumulative=True)  
Print text report  
OUTPUT:
Nothing. A textual report is printed to stdout.  
EXAMPLES:

```
sage: from sage.misc.gperftools import Profiler
sage: prof = Profiler()
sage: prof.start()  # optional - gperftools
sage: # do something
sage: prof.stop()  # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
sage: prof.top()  # optional - gperftools
Using local file ...
Using local file ...
```

`sage.misc.gperftools.crun(s, evaluator)`  
Profile single statement.

- `s` – string. Sage code to profile.
- `evaluator` – callable to evaluate.

EXAMPLES:

```
sage: import sage.misc.gperftools as gperf
sage: ev = lambda ex:eval(ex, globals(), locals())
sage: gperf.crun('gperf.run_100ms()', evaluator=ev)  # optional - gperftools
PROFILE: interrupts/evictions/bytes = ...
Using local file ...
Using local file ...
```
A function that performs some computation for more than (but not that much more than) 100ms.

```
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.
```

```
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:

```
sage: obj = X()
sage: obj.\_doc\_ = "Very special doc"
sage: print(obj.\_doc\_)
Very special doc
```

This normally does not work on extension types:

```
sage: Y().\_doc\_ = "Very special doc"
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:

```
sage: @instancedoc
....: class Meta(type):
....:     "Metaclass doc"
....:     def _instancedoc_(self):
....:         return "Docstring for {}".format(self)
sage: class T(metaclass=Meta):
....:     pass
sage: print(T.\_doc\_)
Docstring for <class '__main__.T'>
sage: class U(metaclass=Meta):
....:     "Special doc for U"
sage: print(U.\_doc\_)
Special doc for U
```

**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:
```python
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.instdoc(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:**

```python
sage: from sage.misc.sageinspect import *
```

Test introspection of modules defined in Python and Cython files:

Cython modules:

```python
sage: sage_getfile(sage.rings.rational)
'.../rational.pyx'
sage: sage_getdoc(sage.rings.rational).lstrip()
'Rational Numbers...'
sage: sage_getsource(sage.rings.rational)
'# distutils: ...Rational Numbers...'
```

Python modules:
Test introspection of classes defined in Python and Cython files:

Cython classes:

```
sage: sage_getfile(sage.rings.rational.Rational)
'../rational.pxd'
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()[:50]
'Provide a tokeneater() method to detect the...
```

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.pxd'
sage: sage_getdoc(sage.rings.rational.make_rational).lstrip()[:40]
'Make a rational number ...
```

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, **kwds)'
sage: sage_getdef(str.find, 'find')
'find(*args, **kwds)'

By trac ticket #9976 and trac ticket #14017, introspection also works for interactively defined Cython code, and with rather tricky argument lines:

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: '_attributes' in attrs and '_fields' in attrs and 'col_offset' in attrs
True
sage: visitor.visit(v.targets[0])
'jc'
sage: visitor.visit(v.value)
['veni', 'vidi', 'vici']

visit_BinOp(node)
Visit a Python AST ast.BinOp node.

INPUT:
• node - the node instance to visit

OUTPUT:
• The result that node represents

AUTHOR:
• Simon King

EXAMPLES:

sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit(ast.parse(x).body[0].value)
sage: [vis(d) for d in ['(3+(2*4))', '7|8', '5^3', '7/3', '7//3', '3<<4']]
→ #indirect doctest
[11, 15, 6, 2.3333333333333335, 2, 48]

visit_BoolOp(node)

INPUT:
• node - the node instance to visit

OUTPUT:
• The result that node represents

AUTHOR:
• Simon King

EXAMPLES:

sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()
sage: vis = lambda x: visitor.visit(ast.parse(x).body[0].value)
sage: [vis(d) for d in ['True and 1', 'False or 3 or None', '3 and 4']]
→ #indirect doctest
[1, 3, 4]

visit_Compare(node)
Visit a Python AST ast.Compare node.


INPUT:
• node - the node instance to visit

OUTPUT:
• The result that node represents

AUTHOR:
• Simon King

EXAMPLES:

```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}']]  
[[], [(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]]]]
[[], ['s', 't', 'u'], [[e'], []], ['pi']]
```

`visit_Name(node)`
Visit a Python AST `ast.Name` node.

INPUT:
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:

```python
sage: import ast, sage.misc.sageinspect as sms
sage: visitor = sms.SageArgSpecVisitor()

sage: vis = lambda x: visitor.visit_Name(ast.parse(x).body[0].value)
sage: [vis(n) for n in ['foo', 'bar']]
['foo', 'bar']
sage: [type(vis(n)) for n in ['foo', 'bar']]
<class 'str'>, <class 'str'>
```

visit_Num\( (node) \)
Visit a Python AST `ast.Num` node.

INPUT:

• node - the node instance to visit

OUTPUT:

• the number the node represents

EXAMPLES:

```python
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']]
[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', 'syntax', r'''r'kee''']]
[['abstract', '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.UnaryOp 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_UnaryOp(ast.parse(x).body[0].value)
sage: [vis(d) for d in ['+(3*2)', '-(3*2)']]
[6, -6]
```

visit_arg(node)
Visit a Python AST ast.arg node.
This node type is only on Python 3, where function arguments are more complex than just an identifier (e.g. they may also include annotations).

For now we simply return the argument identifier as a string.

**INPUT:**
- node – the node instance to visit

**OUTPUT:**
the argument name

**EXAMPLES:**

```python
sage: import ast, sage.misc.sageinspect as sms
sage: s = "def f(a, b=2, c={"a": [4, 5.5, False]}, d=(None, True)):\n      return"

sage: visitor = sms.SageArgSpecVisitor()

sage: args = ast.parse(s).body[0].args.args

sage: [visitor.visit_arg(n) for n in args]
['a', 'b', 'c', 'd']
```

`sage.misc.sageinspect.formatannotation(annotation, base_module=None)`
This is taken from Python 3.7’s inspect.py; the only change is to add documentation.

**INPUT:**
- annotation – annotation for a function
- base_module (optional, default None)

This is only relevant with Python 3, so the doctests are marked accordingly.

**EXAMPLES:**

```python
sage: from sage.misc.sageinspect import formatannotation
sage: import inspect

sage: def foo(a, *, b:int, **kwargs):
    ....: pass

sage: s = inspect.signature(foo)

sage: a = s.parameters['a'].annotation
sage: a
<class 'inspect._empty'>

sage: formatannotation(a)
'inspect._empty'

sage: b = s.parameters['b'].annotation
sage: b
<class 'int'>

sage: formatannotation(b)
'int'
```

`sage.misc.sageinspect.is_function_or_cython_function(obj)`
Check whether something is a function.

This is a variant of `inspect.isfunction()`: We assume that anything which has a genuine `__code__` attribute (not using `__getattr__` overrides) is a function. This is meant to support Cython functions.

Think twice before using this function (or any function from the `inspect` or `sage.misc.sageinspect` modules). Most uses of `inspect.isfunction()` in ordinary library code can be replaced by `callable()`.
EXAMPLES:

```
sage: from sage.misc.sageinspect import is_function_or_cython_function
sage: def f(): pass
sage: is_function_or_cython_function(f)
True
sage: is_function_or_cython_function(lambda x:x)
True
sage: from sage.categories.coercion_methods import _mul_parent
sage: is_function_or_cython_function(_mul_parent)
True
sage: is_function_or_cython_function(Integer.digits)  # unbound method
False
sage: is_function_or_cython_function(Integer(1).digits)  # bound method
False
```

`sage.misc.sageinspect.isclassinstance(obj)`
Check if argument is instance of non built-in class

INPUT: obj – object

EXAMPLES:

```
sage: from sage.misc.sageinspect import isclassinstance
sage: isclassinstance(int)
False
sage: class myclass: pass
sage: isclassinstance(myclass())
True
sage: isclassinstance(myclass)
False
sage: class mymetaclass(type):
    pass
sage: class myclass2(metaclass=mymetaclass):
    pass
sage: isclassinstance(myclass2)
False
```

`sage.misc.sageinspect.loadable_module_extension()`
Return the filename extension of loadable modules, including the dot.

It is ‘.dll’ on cygwin, ‘.so’ otherwise.

EXAMPLES:

```
sage: from sage.misc.sageinspect import loadable_module_extension
sage: sage.structure.sage_object.__file__.endswith(loadable_module_extension())
True
```

`sage.misc.sageinspect.sage_formatargspec` *(args, varargs=None, varkw=None, defaults=None, kwonlyargs=(), kwonlydefaults={}, annotations={}, formatarg=<function formatarg at 0x7fb4975bedc0>, formatvarargs=<function <lambda> at 0x7fb4975bede0>, formatvarkw=<function <lambda> at 0x7fb4975bee50>, formatvalue=<function <lambda> at 0x7fb4975beee0>, formatreturns=<function <lambda> at 0x7fb4975bef70>, formatannotation=<function formatannotation at 0x7fb4975bed30>)*
Format an argument spec from the values returned by getfullargspec.

7.3. Miscellaneous Inspection and Development Tools
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:**

```python
sage: from sage.misc.sageinspect import sage_formatargspec
sage: 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)
```

We now run sage_formatargspec on some functions from the Sage library:

```python
sage: from sage.misc.sageinspect import sage_getargspec
sage: def f(x, y, z=1, t=2, *args, **keywords):
    ....:    pass
sage: sage_getargspec(f)
ArgSpec(args=['x', 'y', 'z', 't'], varargs='args', keywords='keywords', defaults=(1, 2))
```

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:**

```python
sage: from sage.misc.sageinspect import sage_getargspec
sage: sage_getargspec(f)
ArgSpec(args=['x', 'y', 'z', 't'], varargs='args', keywords='keywords', defaults=(1, 2))
```
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=(None, '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', 'None', False))
```

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: 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.

- obj - function
- obj_name - string (optional, default '')

obj_name is prepended to the output.

EXAMPLES:

```
sage: from sage.misc.sageinspect import sage_getdef
sage: sage_getdef(identity_matrix)
'(ring, n=0, sparse=False)'
sage: sage_getdef(identity_matrix, 'identity_matrix')
'identity_matrix(ring, n=0, sparse=False)'
```

Check that trac ticket #6848 has been fixed:

```
sage: sage_getdef(RDF.random_element)
'(min=-1, max=1)'
```

If an exception is generated, None is returned instead and the exception is suppressed.

AUTHORS:
- William Stein
- extensions by Nick Alexander

```
sage.misc.sageinspect.sage_getdoc(obj, obj_name='', embedded=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: from sage.misc.sageinspect import sage_getdoc
sage: sage_getdoc(identity_matrix)[87:124]
'Return the n x n identity matrix over '
... 
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.
```

430 Chapter 7. Development Tools
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
sage: from sage.misc.sageinspect import sage_getdoc_original
```

Here is a class that has its own docstring:

```python
sage: 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
sage: from sage.categories.category import Category
sage: class A(Category):
....:     def __init__(self):
....:         '''The __init__ docstring''

sage: sage_getdoc_original(A)
'The __init__ docstring'
```

```python
sage: class B(Category):
....:     pass

sage: sage_getdoc_original(B)
''
```

Old-style classes are supported:

```python
sage: class OldStyleClass:
....:     def __init__(self):
....:         '''The __init__ docstring''
....:         pass

sage: print(sage_getdoc_original(OldStyleClass))
```

The `__init__` docstring

When there is no `__init__` method, we just get an empty string:

```python
sage: class OldStyleClass:
....:     pass

sage: sage_getdoc_original(OldStyleClass)
''
```

If an instance of a class does not have its own docstring, the docstring of its class results:

```python
sage: 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:

```
sage: from sage.misc.sageinspect import sage_getfile
sage: sage_getfile(sage.rings.rational)[-23:]
'sage/rings/rational.pxd'
sage: sage_getfile(Sq)[-42:]
'sage/commutative/steenrod/steenrod_algebra.py'
```

The following tests against some bugs fixed in trac ticket #9976:

```
sage: obj = sage.combinat.partition_algebra.SetPartitionsAk
sage: sage_getfile(obj)
'...sage/combinat/partition_algebra.py'
```

And here is another bug, fixed in trac ticket #11298:

```
sage: P.<x,y> = QQ[

sage: sage_getfile(P)
'...sage/rings/polynomial/multi_polynomial_libsingular...'
```

A problem fixed in trac ticket #16309:

```
sage: cython('''
      ....: class Bar: pass
      ....: cdef class Foo: pass
      ....: ''

sage: sage_getfile(Bar)
'...pyx'
sage: sage_getfile(Foo)
'...pyx'
```

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?:

```
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

\texttt{sage.misc.sageinspect.sage_getsourcelines(obj)}

Return a pair ([source_lines], starting line number) of the source code associated to \texttt{obj}, or None.

INPUT:

- \texttt{obj} – function, etc.

OUTPUT:

(source_lines, lineno) or None: source_lines is a list of strings, and lineno is an integer.

EXAMPLES:

```
sage: from sage.misc.sageinspect import sage_getsourcelines
sage: sage_getsourcelines(matrix)[1]
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 \texttt{CachedFunction}:

```
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.

\texttt{sage.misc.sageinspect.sage_getvariablename(self, omit_underscore_names=True)}

Attempt to get the name of a Sage object.

INPUT:

- \texttt{self} – any object.
- \texttt{omit_underscore_names} – boolean, default True.
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](http://pythonic.pocoo.org/2009/5/30/finding-objects-names), written by Georg Brandl.

**EXAMPLES:**

```python
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:

```python
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.

```python
sage.misc.edit_module.edit(obj, editor=None, bg=None)
```

Open source code of `obj` in editor of your choice.

**INPUT:**

- `editor` – str (default: None); If given, use specified editor. Choice is stored for next time.

**AUTHOR:**

Nils Bruin (2007-10-03)

**EXAMPLES:**

This is a typical example of how to use this routine:
# make some object obj
sage: edit(obj)  # not tested

Now for more details and customization:

sage: import sage.misc.edit_module as m
sage: m.set_edit_template("vi -c \${line} \${file}\")

In fact, since vi is a well-known editor, you could also just use:

sage: m.set_editor("vi")

To illustrate:

sage: m.edit_template.template
'vi -c ${line} ${file}'

And if your environment variable EDITOR is set to a recognised editor, you would not have to set anything.

To edit the source of an object, just type something like:

sage: edit(edit)  # not tested

sage.misc.edit_module.edit_devel(self, filename, linenum)
This function is for internal use and is called by IPython when you use the IPython commands %edit or %ed.

This hook calls the default implementation, but changes the filename for files that appear to be from the sage library: if the filename begins with ‘SAGE_LOCAL/lib/python.../site-packages’, it replaces this by ‘SAGE_ROOT/src’.

EXAMPLES:

sage: %edit gcd  # indirect doctest, not tested
sage: %ed gcd  # indirect doctest, not tested

The above should open your favorite editor (as stored in the environment variable EDITOR) with the file in which gcd is defined, and when your editor supports it, also at the line in which gcd is defined.

sage.misc.edit_module.file_and_line(obj)
Look up source file and line number of obj.

If the file lies in the Sage library, the path name of the corresponding file in the current branch (i.e., the file that gets copied into the Sage library upon running ‘sage -br’). Note that the first line of a file is considered to be 1 rather than 0 because most editors think that this is the case.

AUTHORS:
• Nils Bruin (2007-10-03)
• Simon King (2011-05): Use sageinspect to get the file and the line.

EXAMPLES:

sage: import sage.misc.edit_module as edit_module
sage: edit_module.file_and_line(sage)
('...sage/__init__.py', 0)

The following tests against a bug that was fixed in trac ticket #11298:
sage: edit_module.file_and_line(x)
('...sage/symbolic/expression...pyx', ...)

sage.misc.edit_module.set_edit_template(template_string)
Set the default edit template string.
It should reference ${file} and ${line}. This routine normally needs to be called prior to using 'edit'. However, if the editor set in the shell variable EDITOR is known, then the system will substitute an appropriate template for you. See edit_module.template_defaults for the recognised templates.

AUTHOR:
Nils Bruin (2007-10-03)

EXAMPLES:

sage: from sage.misc.edit_module import set_edit_template
sage: set_edit_template("echo EDIT ${file}:${line}")
sage: edit(sage)  # 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:

sage: from sage.misc.edit_module import set_editor
sage: set_editor('vi')
sage: sage.misc.edit_module.edit_template.template
'vi -c ${line} ${file}'

sage.misc.edit_module.template_fields(template)
Given a String.Template object, returns the fields.

AUTHOR:
Nils Bruin (2007-10-22)

EXAMPLES:

sage: from sage.misc.edit_module import template_fields
sage: from string import Template
sage: t = Template("Template ${one} with ${two} and ${three}")
sage: sorted(template_fields(t))
['one', 'three', 'two']
7.3.4 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_flat
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', 'Polynomial_padic_flat']
sage: G.edges(labels=False)
[['Polynomial_padic', 'Polynomial'], ['Polynomial_padic_capped_relative_dense', 'Polynomial_generic_cdv'], ['Polynomial_padic_capped_relative_dense', 'Polynomial_padic'], ['Polynomial_padic_flat', 'Polynomial_generic_dense'], ['Polynomial_padic_flat', 'Polynomial_padic']]```

We construct the inheritance graph of a given class:

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

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

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

**Warning:** the output of `class_graph` used to be a dictionary mapping each class name to the list of names of its bases. This can be emulated by setting the option `as_graph` to `False`:
sage: class_graph(sage.rings.polynomial.padics, depth=2, as_graph=False)
{'Polynomial_padic': ['Polynomial'],
 'Polynomial_padic_capped_relative_dense': ['Polynomial_generic_cdv',
 'Polynomial_padic'],
 'Polynomial_padic_flat': ['Polynomial_generic_dense', 'Polynomial_padic']}

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

Note: class_graph does not yet handle nested classes

7.3.5 Some tools for developers

AUTHORS:

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

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

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

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

EXAMPLES:

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

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

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

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

If 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:

```
sage: import sage.misc.dev_tools as dt
sage: dt.find_objects_from_name('FareySymbol')
[[<class 'sage.modular.arithgroup.farey_symbol.Farey'>]]
sage: import sympy
(continues on next page)
Examples that do not belong to the global namespace but in a loaded module:

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

**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:**

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

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

Print import statements for the given objects.

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

```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 objects 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 Z

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

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

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

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

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

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

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

sage: import_statements('cached_function')
from sage.misc.cachefunc import cached_function

sage: import_statements('Z')
# **Warning**: distinct objects with name 'Z' in:
#   - sage.calculus.predefined
#   - sage.rings.integer_ring
from sage.rings.integer_ring import Z
```
Specifying a string is also useful for objects that are not imported in the Sage interpreter namespace by default. In this case, an object with that name is looked up in all the modules that have been imported in this session:

```sage
sage: import_statement_string
Traceback (most recent call last):
... NameError: name 'import_statement_string' is not defined
sage: import_statements("import_statement_string")
from sage.misc.dev_tools import import_statement_string
```

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

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

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

```sage
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
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
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__,
                deprecated=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__, deprecated=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.cif import CIF
sage: import_statements(NaN)
from sage.symbolic.constants import NaN
sage: import_statements(pi)
from sage.symbolic.constants import pi
sage: import_statements('SAGE_ENV')
from sage.env import SAGE_ENV
sage: import_statements('graph_decompositions')
import sage.graphs.graph_decompositions

Check that a name from the global namespace is properly found (see trac ticket #23779):

sage: import_statements('log')
from sage.misc.functional 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 modules.

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

INPUT:

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

EXAMPLES:

sage: sage.misc.dev_tools.load_submodules(sage.combinat)
load sage.combinat.algebraic_combinatorics... succeeded
...
load sage.combinat.words.suffix_trees... succeeded
Calling a second time has no effect (since the function does not import modules already imported):

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

The second argument allows to exclude a pattern:

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

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

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

Graphical profiling with runsnake

**INPUT:**

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

**EXAMPLES:**

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

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

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

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

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

See the runsnake website for instructions for other platforms.

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

**See also:**

- The runsnake website
- `%prun`
- `Profiler`
7.3.6 Function Mangling

This module provides utilities for extracting information about python functions.

AUTHORS:
- Tom Boothby (2009): Original version in 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,

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

the following calls are equivalent,

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

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

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

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

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:

```
sage: from sage.misc.function_mangling import ArgumentFixer
sage: def wrap2(g):
....:     af = ArgumentFixer(g)
```

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

    def _g(*args, **kwargs):
        print(af.fix_to_pos())
        return g(*args, **kwargs)
    return _g

sage: h2 = wrap2(f)
sage: t = h2()  
((10,), ())
sage: t = h2(10)  
((10,), ())
sage: t = h2(x=10)  
((10,), ())

.. code-block:: python

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

f

fix_to_named(*args, **kwargs)

Normalize the arguments with a preference for named arguments.

INPUT:

* any positional and named arguments.

OUTPUT:

We return a tuple

\[(e_1, e_2, ..., e_k), ((n_1, v_1), ..., (n_m, v_m))\]

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

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

\[
A,K = self.fix_to_pos(...)
self.f(*A,**dict(K))
\]

and

\[
self.f(...)
\]

in all cases.

EXAMPLES:

.. code-block:: python

    from sage.misc.function_mangling import ArgumentFixer
    def sum3(a,b,c=3,*args,**kwargs):
        return a+b+c
    AF = ArgumentFixer(sum3)
fix_to_pos(*args, **kwds)

Normalize the arguments with a preference for positional arguments.

INPUT:
Any positional or named arguments

OUTPUT:
We return a tuple

\[(e_1, e_2, ..., e_k), ((n_1, v_1), ..., (n_m, v_m))\]

where \(n_1, ..., n_m\) are the names of the arguments and \(v_1, ..., v_m\) are the values passed in; and \(e_1, ..., e_k\) are

the unnamed arguments. We minimize \(m\).

The commands

\[
\begin{align*}
A, K &= \text{self}.\text{fix}_\text{to}_\text{pos}(\ldots) \\
\text{self}.f(*A, **\text{dict}(K))
\end{align*}
\]

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 \(n_1, ..., n_m\) are given in alphabetical order.

EXAMPLES:

```python
sage: from sage.misc.function_mangling import ArgumentFixer
dsage: def do_something(a, b, c=3, *args, **kwargs):
      print("{} {} {} {} {}\n      .....: print("{} {} {} {} {}\n      .....: sorted(kwarg.items()))
dsage: AF = ArgumentFixer(do_something)
dsage: A, K = AF.fix_to_pos(1, 2, 3, 4, 5, 6, f=14, e=16)
dsage: print("{} {} {} {} {}\n      (1, 2, 3, 4, 5, 6) [('e', 16), ('f', 14)]
dsage: do_something('A, **\text{dict}(K))
1 2 3 (4, 5, 6) [('e', 16), ('f', 14)]
dsage: do_something(1, 2, 3, 4, 5, 6, f=14, e=16)
(1, 2, 3, 4, 5, 6) [('e', 16), ('f', 14)]
```
7.3.7 ReST index of functions

This module contains a function that generates a ReST index table of functions for use in doc-strings.

| 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
def a():
    print("Hey")
```

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:

```sage
from sage.misc.rest_index_of_methods import gen_rest_table_index
g = gen_rest_table_index([graphs.PetersenGraph])
```

(continues on next page)
The table of a module:

```python
sage: print(gen_rest_table_index(sage.misc.rest_index_of_methods))
.. csv-table::
   :class: contentstable
   :widths: 30, 70
   :delim: @

   :func:`~sage.misc.rest_index_of_methods.doc_index` @ Attribute an index name to a function.
   :func:`~sage.misc.rest_index_of_methods.gen_rest_table_index` @ Return a ReST table describing a list of functions.
   :func:`~sage.misc.rest_index_of_methods.gen_thematic_rest_table_index` @ Return a ReST string of thematically sorted function (or methods) of a module (or class).
   :func:`~sage.misc.rest_index_of_methods.list_of_subfunctions` @ Returns the functions (resp. methods) of a given module (resp. class) with their names.
```

The table of a class:

```python
sage: print(gen_rest_table_index(Graph))
.. csv-table::
   :class: contentstable
   :widths: 30, 70
   :delim: @

   ...:meth:`~sage.graphs.graph.Graph.sparse6_string` @ Return the sparse6 representation of the graph as an ASCII string.
   ...
```

```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]
```

(continues on next page)
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:

```python
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:


`sage.misc.c3.C3_algorithm(start, bases, attribute, proper)`
An implementation of the C3 algorithm.

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

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

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

INPUT:

- `start` – an object; the returned list is built upon data provided by certain attributes of `start`. 

• bases – a string; the name of an attribute of start providing a list of objects.

• attribute – a string; the name of an attribute of the objects provided in getattr(start, bases). That attribute is supposed to provide a list.

ASSUMPTIONS:

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

OUTPUT:

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

EXAMPLES:

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

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

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

We construct a little hierarchy:

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

And inspect the linear extensions associated to each element:

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

So far so good. However:

```
sage: Foo._all_bases
Traceback (most recent call last):
  ...
ValueError: 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 Foo, since A and B impose incompatible orders. If the above was a hierarchy of classes, Python would complain that it cannot calculate a consistent Method Resolution Order.

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

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

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

```python
sage: a = A7()
sage: a.foo()
4
```
Specifically, the MRO is calculated using the so-called C3 algorithm which guarantees that the MRO respects not only inheritance, but also the order in which the bases (direct super classes) are given for each class.

However, for large hierarchies of classes with lots of multiple inheritance, like those derived from categories in Sage, this algorithm easily fails if the order of the bases is not chosen consistently (here for $A_2$ w.r.t. $A_1$):

```python
sage: class B6(A1,A2): pass
sage: class B7(B6,A5): pass
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 dir command in Python has no cache, and its complexity is linear in the number of maximal paths in the class hierarchy graph as defined by the bases. For a linear hierarchy, this is of complexity $O(p_n)$ where $p_n$ is the number of integer partitions of $n$, which is exponential. And indeed, running dir for a typical class like GradedHopfAlgebrasWithBasis(QQ).parent_class with 37 super classes took 18 seconds with this approach.

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

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

Then, running the standard C3 algorithm with the updated list of bases (which we call controlled C3) yields the desired MRO.

EXAMPLES:

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

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

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

And build a HierarchyElement from it:

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

Here are its bases:

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

Using the standard C3 algorithm fails:

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

Now we play with a specific linear extension of $P$:

```python
sage: Q = P.linear_extension([10, 9, 8, 7, 6, 5, 4, 1, 2, 3]).to_poset()
sage: Q.cover_relations()
[[10, 9], [10, 8], [10, 7], [9, 6], [9, 3], [8, 5], [8, 2], [7, 4], [7, 1], [6, 2], [6, 1], [5, 3], [5, 1], [4, 3], [4, 2]]
```

```python
sage: x = HierarchyElement(10, Q)
sage: 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
sage: x.mro
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
sage: x.mro_controlled
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

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

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

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

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

Altogether, four bases were added for control:

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

This information can also be recovered with:

```python
sage: x.all_bases_len()
15
```

(continues on next page)
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:

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

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

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

We now consider a hierarchy of categories:

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

```python
sage: C = FiniteFields()
sage: x = HierarchyElement(C, attrcall("super_categories"), attrgetter("_cmp_key"))
sage: x.mro == x.mro_standard
False
sage: x.all_bases_len()
70
sage: x.all_bases_controlled_len()
74
```

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

```python
sage: C = GradedHopfAlgebrasWithBasis(QQ)
```
sage: x = HierarchyElement(C, attrcall("super_categories"), attrgetter("_cmp_key"))
sage: x._test_mro()
False
sage: x.mro == x.mro_standard
False
sage: x.all_bases_len()
114
sage: x.all_bases_controlled_len()
117

The following can be used to search through the Sage named categories for any that requires the addition of some bases. The output may change a bit when the category hierarchy is changed. As long as the list below does not change radically, it’s fine to just update this doctest:

```python
sage: from sage.categories.category import category_sample
sage: sorted([C for C in category_sample() 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:

```python
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, \texttt{C3\_merge()} returns right away a sorted list:


\begin{verbatim}
sage: from sage.misc.c3_controlled import C3_merge
sage: C3_merge([[2],[1]])
[2, 1]
\end{verbatim}

In that case, \texttt{C3\_sorted\_merge()} returns the same result, with the last line unchanged:


\begin{verbatim}
sage: from sage.misc.c3_controlled import C3_sorted_merge
sage: C3_sorted_merge([[2],[1]])
([2, 1], [1])
\end{verbatim}

On the other hand, with the following input, \texttt{C3\_merge()} returns a non sorted list:


\begin{verbatim}
sage: C3_merge([[1],[2]])
[1, 2]
\end{verbatim}

Then, \texttt{C3\_sorted\_merge()} returns a sorted list, and suggests to replace the last line by [2, 1]:


\begin{verbatim}
sage: C3_sorted_merge([[1],[2]])
([2, 1], [2, 1])
\end{verbatim}

And indeed \texttt{C3\_merge()} now returns the desired result:


\begin{verbatim}
sage: C3_merge([[1],[2,1]])
[2, 1]
\end{verbatim}

From now on, we use this little wrapper that checks that \texttt{C3\_merge()}, with the suggestion of \texttt{C3\_sorted\_merge()}, returns a sorted list:


\begin{verbatim}
sage: def C3_sorted_merge_check(lists):
    ....:     result, suggestion = C3_sorted_merge(lists)
    ....:     assert result == C3_merge(lists[:-1] + [suggestion])
    ....:     return result, suggestion
\end{verbatim}

Base cases:


\begin{verbatim}
sage: C3_sorted_merge_check([])
Traceback (most recent call last):
...
ValueError: The input should be a non empty list of lists (or iterables)
sage: C3_sorted_merge_check([[]])
([], [])
sage: C3_sorted_merge_check([[1]])
([1], [1])
sage: C3_sorted_merge_check([[3,2,1]])
([3, 2, 1], [3, 2, 1])
sage: C3_sorted_merge_check([[1],[1]])
([1], [1])
sage: C3_sorted_merge_check([[3,2,1],[3,2,1]])
([3, 2, 1], [3, 2, 1])
\end{verbatim}
Exercise different states for the last line:

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

Explore (all?) the different execution branches:

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

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

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

```python
class sage.misc.c3_controlled.CmpKey
Bases: object

This class implements the lazy attribute Category._cmp_key.

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

The comparison key should satisfy the following properties:

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

The rationale for the inversion above between $A < B$ and $A \_\text{cmp\_key} > B \_\text{cmp\_key}$ is that we want the order to be compatible with inclusion of categories, yet it’s easier in practice to create keys that get bigger and bigger while we go down the category hierarchy.

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

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

**Todo:**

• Handle nicely (covariant) functorial constructions and axioms

**EXAMPLES:**

```python
sage: Objects\_\_cmp\_key
(0, 0)
sage: SetsWithPartialMaps\_\_cmp\_key
(0, 1)
sage: Sets\_\_cmp\_key
(0, 2)
sage: Sets\_Facade\_\_cmp\_key
(1, ...)
sage: Sets\_\_Finite\_\_cmp\_key
(2, ...)
sage: Sets\_\_Infinite\_\_cmp\_key
(4, ...)
sage: EnumeratedSets\_\_cmp\_key
(8, ...)
sage: FiniteEnumeratedSets\_\_cmp\_key
(10, ...)
sage: SetsWithGrading\_\_cmp\_key
(16, ...)
sage: Posets\_\_cmp\_key
(32, ...)
sage: LatticePosets\_\_cmp\_key
(96, ...)
sage: Crystals\_\_cmp\_key
(136, ...)
sage: AdditiveMagmas\_\_cmp\_key
(256, ...)
sage: Magmas\_\_cmp\_key
(4096, ...)
sage: CommutativeAdditiveSemigroups\_\_cmp\_key
```

(continues on next page)
For backward compatibility we currently want the following comparisons:

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

```python
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.

EXAMLES:

```python
sage: Algebras(GF(3))._cmp_key == Algebras(GF(5))._cmp_key  # indirect doctest
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:

```
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],
    ....: 37 : [30, 29, 28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
    ....: 36 : [35, 30, 29, 28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
    ....: 35 : [29, 28, 27, 26, 15, 14, 12, 11, 9, 8, 5, 3, 2, 1, 0],
    ....: 34 : [33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17,
    ....: 16, 15, 14, 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
<class '44.cls'>
sage: x.cls.mro()
[<class '44.cls'>, <class '43.cls'>, <class '42.cls'>, <class '41.cls'>, <class '40.
 → cls'>, <class '39.cls'>, <class '38.cls'>, <class '37.cls'>, <class '36.cls'>,
 → <class '35.cls'>, <class '34.cls'>, <class '33.cls'>, <class '32.cls'>, <class
 → '31.cls'>, <class '30.cls'>, <class '29.cls'>, <class '28.cls'>, <class
 → '27.cls'>, <class '26.cls'>, <class '25.cls'>, <class '24.cls'>, <class '23.cls'>, <class
 → '22.cls'>, <class '21.cls'>, <class '20.cls'>, <class '19.cls'>, <class '18.cls'>,
 → <class '17.cls'>, <class '16.cls'>, <class '15.cls'>, <class '14.cls'>, <class
 → '13.cls'>, <class '12.cls'>, <class '11.cls'>, <class '10.cls'>, <class '9.cls'>,
 → <class '8.cls'>, <class '7.cls'>, <class '6.cls'>, <class '5.cls'>, <class '4.cls'
 → >, <class '3.cls'>, <class '2.cls'>, <class '1.cls'>, <class '0.cls'>, <...
**all_bases()**

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

```
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.

```
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.

```
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.

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

Return a Python class with inheritance graph parallel to the hierarchy above \texttt{self}.

\begin{verbatim}
  sage: from sage.misc.c3_controlled import HierarchyElement
  sage: P = Poset((divisors(30), lambda x,y: y.divides(x)), facade=True)
  sage: x = HierarchyElement(1, P)
  sage: x.cls
  <class '1.cls'>
  sage: x.cls.mro()
  [<class '1.cls'>, <... 'object'>]
  sage: x = HierarchyElement(30, P)
  sage: x.cls
  <class '30.cls'>
  sage: x.cls.mro()
  [<class '30.cls'>, <class '15.cls'>, <class '10.cls'>, <class '6.cls'>, <class '5.cls'>, <class '3.cls'>, <class '2.cls'>, <class '1.cls'>, <... 'object'>]
\end{verbatim}

\texttt{mro()}

The MRO for this object, calculated with \texttt{C3_sorted_merge()}.

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

\texttt{mro_controlled()}

The MRO for this object, calculated with \texttt{C3_merge()}, under control of \texttt{C3_sorted_merge()}

\begin{verbatim}
  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
  (continues on next page)
\end{verbatim}
[7, 6, 5, 4, 3, 2, 1]
sage: x._bases
[6, 5]
sage: x._bases_controlled
[6, 5, 4]
sage: C3_merge([[6, 4, 3], [5, 2, 1], [6, 5]])
[6, 4, 3, 5, 2, 1]
sage: C3_merge([[6, 4, 3], [5, 2, 1], [6, 5, 4]])
[6, 5, 4, 3, 2, 1]

def mro_standard()
    The MRO for this object, calculated with \texttt{C3\_merge()}.

EXAMPLES:

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

sage.misc.c3_controlled.identity(x)

EXAMPLES:

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

NINE

INDICES AND TABLES

• Index
• Module Index
• Search Page
PYTHON MODULE INDEX

a
sage.arith.srange, 83

sage.docs.instancedoc, 417

e
sage.ext.fast_callable, 206
sage.ext.fast_eval, 224
sage.ext.memory, 451

f
sage.features, 225
sage.features.all, 232
sage.features.bliss, 235
sage.features.csdp, 235
sage.features.databases, 236
sage.features.dvipng, 237
sage.features.fes, 237
sage.features.ffmpeg, 237
sage.features.four_ti_2, 238
sage.features.gap, 238
sage.features.graph_generators, 238
sage.features.graphviz, 240
sage.features.imagemagick, 240
sage.features.interfaces, 241
sage.features.internet, 243
sage.features.join_feature, 231
sage.features.kenzo, 243
sage.features.latex, 243
sage.features.latte, 245
sage.features.lrs, 246
sage.features.mcd, 246
sage.features.mctape, 246
sage.features.mip_backends, 247
sage.features.normaliz, 247
sage.features.pandoc, 247
sage.features.pdf2svg, 248
sage.features.pkg_systems, 234
sage.features.polymake, 248
sage.features.rubiks, 248
sage.features.sagemath, 232
sage.features.tdlib, 250

M
sage.media.wav, 119
sage.misc.abstract_method, 53
sage.misc.banner, 369
sage.misc.benchmark, 407
sage.misc.bindable_class, 56
sage.misc.c3, 451
sage.misc.c3_controlled, 453
sage.misc.cachefunc, 171
sage.misc.call, 71
sage.misc.callable_dict, 72
sage.misc.citation, 394
sage.misc.classcall_metaclass, 64
sage.misc.classgraph, 437
sage.misc.constant_function, 64
sage.misc.converting_dict, 73
sage.misc.copying, 395
sage.misc.cython, 251
sage.misc.decorators, 58
sage.misc.defaults, 1
sage.misc.dev_tools, 438
sage.misc.dist, 393
sage.misc.edit_module, 434
sage.misc.element_with_label, 87
sage.misc.explain_pickle, 349
sage.misc.fast_methods, 69
sage.misc.flatten, 75
sage.misc.fpickle, 348
sage.misc.func_persist, 337
sage.misc.function_mangling, 444
sage.misc.functional, 2
sage.misc.gperftools, 414
sage.misc.html, 310
sage.misc.inherit_comparison, 67
sage.misc.inline_fortran, 254
sage.misc.latex, 315
sage.misc.latex_macros, 334
sage.misc.lazy_attribute, 149
sage.misc.lazy_format, 158
sage.misc.lazy_import, 159
sage.misc.lazy_import_cache, 163
sage.misc.lazy_list, 164
sage.misc.lazy_string, 170
sage.misc.map_threaded, 83
sage.misc.mathml, 314
sage.misc.messaging, 93
sage.misc.method_decorator, 68
sage.misc.misc, 126
sage.misc.misc_c, 140
sage.misc.mrange, 77
sage.misc.multireplace, 82
sage.misc.nested_class, 361
sage.misc.nested_class_test, 407
sage.misc.object_multiplexer, 69
sage.misc.package, 386
sage.misc.pager, 376
sage.misc.parser, 255
sage.misc.persist, 337
sage.misc.prandom, 44
sage.misc.profiler, 413
sage.misc.python, 264
sage.misc.random_testing, 404
sage.misc.randstate, 33
sage.misc.remote_file, 93
sage.misc.repr, 283
sage.misc.reset, 371
sage.misc.rest_index_of_methods, 447
sage.misc.sage_eval, 265
sage.misc.sage_input, 285
sage.misc.sage_ostools, 94
sage.misc.sage_timeit, 410
sage.misc.sage_timeit_class, 412
sage.misc.sage_unittest, 397
sage.misc.sagedoc, 376
sage.misc.sageinspect, 419
sage.misc.search, 76
sage.misc.session, 365
sage.misc.sh, 268
sage.misc.sphinxify, 385
sage.misc.stopgap, 122
sage.misc.superseded, 123
sage.misc.table, 305
sage.misc.temperature_file, 88
sage.misc.unknown, 50
sage.misc.verbose, 146
sage.misc.viewer, 373
sage.misc.weak_dict, 197

S
sage.structure.graphics_file, 118

t
sage.typeset.ascii_art, 277
Symbols

_rich_repr_() (sage.misc.html.HtmlFragment method), 312
_rich_repr_() (sage.misc.table.table method), 309

A

abs() (sage.ext.fast_callable.Expression method), 211
absolute_path() (sage.features.latex.TeXFile method), 244
absolute_path() (sage.features.StaticFile method), 230
abstract_method() (in module sage.misc.abstract_method), 53
abstract_methods_of_class() (in module sage.misc.abstract_method), 56
AbstractMethod (class in sage.misc.abstract_method), 53
add_column() (sage.databases.sql_db.SQLDatabase method), 100
add_data() (sage.databases.sql_db.SQLDatabase method), 101
add_library() (sage.misc.inline_fortran.InlineFortran method), 254
add_library_path() (sage.misc.inline_fortran.InlineFortran method), 254
add_macro() (sage.misc.latex.Latex method), 316
add_package_to_preamble_if_available() (sage.misc.latex.Latex method), 316
add_row() (sage.databases.sql_db.SQLDatabase method), 101
add_rows() (sage.databases.sql_db.SQLDatabase method), 102
add_to_preamble() (sage.misc.latex.Latex method), 316
additive_order() (in module sage.misc.functional), 5
all_bases() (sage.misc.c3_controlled.HierarchyElement method), 465
all_bases_controlled_len() (sage.misc.c3_controlled.HierarchyElement method), 465
all_bases_len() (sage.misc.c3_controlled.HierarchyElement method), 465
all_features() (in module sage.features.all), 232
all_features() (in module sage.features.bliss), 235
all_features() (in module sage.features.csdp), 235
all_features() (in module sage.features.databases), 236
all_features() (in module sage.features.dvipng), 237
all_features() (in module sage.features.ffmpeg), 238
all_features() (in module sage.features.four_ti_2), 238
all_features() (in module sage.features.graph_generators), 239
all_features() (in module sage.features.graphviz), 240
all_features() (in module sage.features.imagemagick), 241
all_features() (in module sage.features.interfaces), 242
all_features() (in module sage.features.internet), 243
all_features() (in module sage.features.kenzo), 243
all_features() (in module sage.features.latex), 244
all_features() (in module sage.features.latte), 245
all_features() (in module sage.features.lrs), 246
all_features() (in module sage.features.mip_backends), 246
all_features() (in module sage.features.mip_backends), 247
all_features() (in module sage.features.normaliz), 247
all_features() (in module sage.features.pdf2svg), 248
all_features() (in module sage.features.polymake), 248
all_features() (in module sage.features.rubiks), 248
all_features() (in module sage.features.sagemath), 232
all_features() (in module sage.features.tdlib), 250
APPEND() (sage.misc.explain_pickle.PickleExplainer method), 351
append() (sage.misc.explain_pickle.TestAppendList method), 356
APPENDS() (sage.misc.explain_pickle.PickleExplainer method), 352
ArgumentFixer (class in sage.misc.function_mangling), 444
arguments (sage.ext.fast_callable.ExpressionCall method), 212
ascii_art (in module sage.typeset.ascii_art), 279
AsciiArt (class in sage.typeset.ascii_art), 279
assign (sage.misc.sage_input.SageInputBuilder method), 292
atomic_dir (in module sage.misc.temporary_file), 88
atomic_write (in module sage.misc.temporary_file), 88
attrcall (in module sage.misc.call), 71
AttrCallObject (class in sage.misc.call), 71

B
BackslashOperator (class in sage.misc.misc), 126
backtrack (sage.misc.parser.Tokenizer method), 261
balanced_sum (in module sage.misc.misc_c), 140
banner (in module sage.misc.banner), 369
banner_text (in module sage.misc.banner), 369
base (sage.ext.fast_callable.ExpressionIPow method), 214
base_field (in module sage.misc.functional), 5
base_ring (in module sage.misc.functional), 5
bases (sage.misc.c3_controlled.HierarchyElement method), 465
basis (in module sage.misc.functional), 5
bench0 (in module sage.misc.benchmark), 407
bench1 (in module sage.misc.benchmark), 407
bench2 (in module sage.misc.benchmark), 408
bench3 (in module sage.misc.benchmark), 408
bench4 (in module sage.misc.benchmark), 408
bench5 (in module sage.misc.benchmark), 408
bench6 (in module sage.misc.benchmark), 408
bench7 (in module sage.misc.benchmark), 408
benchmark (in module sage.misc.benchmark), 409
benchmark_libc (in module sage.misc.randstate), 40
benchmark_mtt (in module sage.misc.randstate), 40
Benzene (class in sage.features.graph_generators), 238
betavariate (in module sage.misc.prandom), 45
BindableClass (class in sage.misc.bindable_class), 56
BINFLOAT (sage.misc.explain_pickle.PickleExplainer method), 352
BINGET (sage.misc.explain_pickle.PickleExplainer method), 352

BlackslashOperator (class in sage.misc.misc), 126
backtrack (sage.misc.parser.Tokenizer method), 261
balanced_sum (in module sage.misc.misc_c), 140
banner (in module sage.misc.banner), 369
banner_text (in module sage.misc.banner), 369
base (sage.ext.fast_callable.ExpressionIPow method), 214
base_field (in module sage.misc.functional), 5
base_ring (in module sage.misc.functional), 5
bases (sage.misc.c3_controlled.HierarchyElement method), 465
basis (in module sage.misc.functional), 5
bench0 (in module sage.misc.benchmark), 407
bench1 (in module sage.misc.benchmark), 407
bench2 (in module sage.misc.benchmark), 408
bench3 (in module sage.misc.benchmark), 408
bench4 (in module sage.misc.benchmark), 408
bench5 (in module sage.misc.benchmark), 408
bench6 (in module sage.misc.benchmark), 408
bench7 (in module sage.misc.benchmark), 408
benchmark (in module sage.misc.benchmark), 409
benchmark_libc (in module sage.misc.randstate), 40
benchmark_mtt (in module sage.misc.randstate), 40
Benzene (class in sage.features.graph_generators), 238
betavariate (in module sage.misc.prandom), 45
BindableClass (class in sage.misc.bindable_class), 56
BINFLOAT (sage.misc.explain_pickle.PickleExplainer method), 352
BINGET (sage.misc.explain_pickle.PickleExplainer method), 352

BSTRING (sage.misc.explain_pickle.PickleExplainer method), 352
BINUNICODE (sage.misc.explain_pickle.PickleExplainer method), 352
blackboard_bold (sage.misc.latex.Latex method), 317
Bliss (class in sage.features.bliss), 235
BlissLibrary (class in sage.features.bliss), 235
BlockFinder (class in sage.misc.sageinspect), 421
bool_function (in module sage.misc.function_mangling), 324
bool_function (in module sage.misc.mathml), 314
BoundClass (class in sage.misc.bindable_class), 58
browser (in module sage.misc.viewer), 374
browser (sage.misc.viewer.Viewer method), 373
Buckygen (class in sage.features.graph_generators), 239
BUILD (sage.misc.explain_pickle.PickleExplainer method), 352
build (sage.misc.sage_input.SageInputBuilder method), 212
build_container (sage.misc.sage_input.SageInputBuilder method), 274
build_dict (sage.misc.sage_input.SageInputBuilder method), 275
build_empty (sage.misc.sage_input.SageInputBuilder method), 275
build_from_magic_method (sage.misc.sage_input.SageInputBuilder method), 275
build_from_string (sage.misc.sage_input.SageInputBuilder method), 275
build_list (sage.misc.sage_input.SageInputBuilder method), 275
build_set (sage.misc.sage_input.SageInputBuilder method), 276
build_tuple (sage.misc.sage_input.SageInputBuilder method), 276
builtin_constant_function (in module sage.misc.functional), 325
by_opcode (sage.ext.fast_callable.InterpreterMetadata attribute), 219
by_opname (sage.ext.fast_callable.InterpreterMetadata attribute), 219

C
C3_algorithm (in module sage.misc.c3), 451
C3_merge (in module sage.misc.c3_controlled), 458
C3_sorted_merge (in module sage.misc.c3_controlled), 458
c_rand_double (sage.misc.randstate.randstate method), 41
c_random (sage.misc.randstate.randstate method), 41
cache (sage.misc.cachefunc.CachedFunction attribute), 179
cu2 (class in sage.features.rubiks), 248


cubex (class in sage.features.rubiks), 248

current_op_list() (sage.ext.fast_callable.InstructionStream method), 216

current_randstate() (in module sage.misc.randstate), 40

cyclotomic_polynomial() (in module sage.misc.functional), 7
cyflush() (in module sage.misc.misc_c), 141
cython() (in module sage.misc.cython), 251
cython_compile() (in module sage.misc.cython), 252
cython_import() (in module sage.misc.cython), 252
cython_import_all() (in module sage.misc.cython), 252
cython_lambda() (in module sage.misc.cython), 252
cython_profile_enabled() (in module sage.misc.cython), 252

CythonFeature (class in sage.features), 225

D
data() (sage.structure.graphics_file.GraphicsFile method), 118

DatabaseConwayPolynomials (class in sage.features.databases), 236

DatabaseCremona (class in sage.features.databases), 236

DatabaseJones (class in sage.features.databases), 236

DatabaseKnotInfo (class in sage.features.databases), 236
db() (in module sage.misc.persist), 340
db_save() (in module sage.misc.persist), 340
decomposition() (in module sage.misc.functional), 7
decorator_defaults() (in module sage.misc.decorators), 58
decorator_keywords() (in module sage.misc.decorators), 59
default_viewer() (in module sage.misc.viewer), 374
delete_rows() (sage.databases.sql_db.SQLDatabase method), 103
delete_tmpfiles() (in module sage.misc.temporary_file), 91
denominator() (in module sage.misc.functional), 8
deprecated_function_alias() (in module sage.misc.superseded), 123

DeprecationFunctionAlias (class in sage.misc.superseded), 123
deprecation() (in module sage.misc.superseded), 124
deprecation_cython() (in module sage.misc.superseded), 124
dest_fd (sage.misc.sage_ostools.redirection attribute), 96
dest_file (sage.misc.sage_ostools.redirection attribute), 96
det() (in module sage.misc.functional), 8
detex() (in module sage.misc.sagedoc), 376

DICT() (sage.misc.explain_pickle.PickleExplainer method), 352
dict() (sage.misc.sage_input.SageInputBuilder method), 293
dict_function() (in module sage.misc.latex), 325
dict_key() (in module sage.misc.cachefunc), 195
dijkstra() (in module sage.features.rubiks), 249
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), 196

DiskCachedFunction (class in sage.misc.cachefunc), 188
doc_index() (in module sage.misc.rest_index_of_methods), 447
domain() (sage.media.wav.Wave method), 120
dot (in module sage.features.graphviz), 240
drop_column() (sage.databases.sql_db.SQLDatabase method), 104
drop_data_from_table() (sage.databases.sql_db.SQLDatabase method), 104
drop_index() (sage.databases.sql_db.SQLDatabase method), 105
drop_primary_key() (sage.databases.sql_db.SQLDatabase method), 105
drop_table() (sage.databases.sql_db.SQLDatabase method), 106
dummy() (sage.misc.nested_class.MainClass.NestedClass.NestedSubClass method), 365
dumps() (in module sage.misc.persist), 340
dumps() (sage.misc.persist.SagePickle class method), 338

dup() (in module sage.misc.sagedoc), 352
dup_source_fd (sage.misc.sage_ostools.redirection attribute), 96
dvi_viewer() (in module sage.misc.viewer), 375
dvi_viewer() (sage.misc.viewer.Viewer method), 373
dvipng (in module sage.features.dvipng), 237

E
edit() (in module sage.misc.edit_module), 434
edit_devel() (in module sage.misc.edit_module), 435

EDITOR, 434–436

ElementWithLabel (class in sage.misc.element_with_label), 87

ellipses_jiter() (in module sage.arith.srange), 83

ellipses_range() (in module sage.arith.srange), 84
empty() (sage.typeset.character_art.CharacterArt class method), 272
EMPTY_DICT() (sage.misc.explain_pickle.PickleExplainer method), 352
EMPTY_LIST() (sage.misc.explain_pickle.PickleExplainer method), 352
empty_subscript() (sage.misc.sage_input.SageInputBuilder method), 294
EMPTY_TUPLE() (sage.misc.explain_pickle.PickleExplainer method), 352
EmptyNewstyleClass (class in sage.misc.explain_pickle), 351
EmptyOldstyleClass (class in sage.misc.explain_pickle), 351
engine() (sage.misc.latex.Latex method), 317
ensure_startup_finished() (in module sage.misc.lazy_import), 160
environment variable
EDITOR, 434–436
PATH, 94
eta() (in module sage.misc.functional), 9
eval() (sage.misc.html.HTMLFragmentFactory method), 310
eval() (sage.misc.html.MathJax method), 312
eval() (sage.misc.inline_fortran.InlineFortran method), 254
eval() (sage.misc.latex.Latex method), 318
eval() (sage.misc.python.Python method), 264
eval() (sage.misc.sage_timeit_class.SageTimeit method), 413
eval() (sage.misc.sh.Sh method), 268
exactly_one_is_true() (in module sage.misc), 129
Executable (class in sage.features), 226
exists() (in module sage.misc.misc), 129
experimental (class in sage.misc.superseded), 124
experimental_packages() (in module sage.misc.package), 387
experimental_warning() (in module sage.misc.superseded), 125
explain_pickle() (in module sage.misc.explain_pickle), 358
explain_pickle_string() (in module sage.misc.explain_pickle), 359
exponent() (sage.ext.fast_callable.ExpressionIPow method), 214
expovariate() (in module sage.misc.prandom), 45
Expression (class in sage.ext.fast_callable), 211
ExpressionCall (class in sage.ext.fast_callable), 212
ExpressionChoice (class in sage.ext.fast_callable), 212
ExpressionConstant (class in sage.ext.fast_callable), 213
ExpressionIPow (class in sage.ext.fast_callable), 213
ExpressionTreeBuilder (class in sage.ext.fast_callable), 214
ExpressionVariable (class in sage.ext.fast_callable), 215
EXT1() (sage.misc.explain_pickle.PickleExplainer method), 352
EXT2() (sage.misc.explain_pickle.PickleExplainer method), 352
EXT4() (sage.misc.explain_pickle.PickleExplainer method), 352
extend() (sage.misc.explain_pickle.TestAppendList method), 356
extension() (sage.structure.graphics_file.Mime class method), 118
extra_macros() (sage.misc.latex.Latex method), 319
extra_preamble() (sage.misc.latex.Latex method), 319
f (sage.misc.cachefunc.CachedFunction attribute), 179
f (sage.misc.function_mangling.ArgumentFixer attribute), 445
fast_callable() (in module sage.ext.fast_callable), 220
fast_float() (in module sage.ext.fast_eval), 224
FastHashable_class (class in sage.misc.fast_methods), 69
fcp() (in module sage.misc.functional), 9
Feature (class in sage.features), 227
FeatureNotPresentError, 228
FeatureTestResult (class in sage.features), 229
FFmpeg (class in sage.features.ffmpeg), 237
file_and_line() (in module sage.misc.edit_module), 435
file_list() (sage.misc.cachefunc.FileCache method), 189
FileCache (class in sage.misc.cachefunc), 188
filename() (sage.misc.gperftools.Profiler method), 415
filename() (sage.structure.graphics_file/GraphicsFile method), 118
find_object_modules() (in module sage.misc.dev_tools), 438
find_objects_from_name() (in module sage.misc.dev_tools), 438
finish_startup() (in module sage.misc.lazy_import), 160
fix_to_named() (sage.misc.function_mangling.ArgumentFixer method), 445
fix_to_pos() (sage.misc.function_mangling.ArgumentFixer method), 446
flatten() (in module sage.misc.flatten), 75
FLOAT() (sage.misc.explain_pickle.PickleExplainer method), 352
float_function() (in module sage.misc.latex), 326
float_str() (sage.misc.sage_input.SageInputBuilder method), 294
foo() (in module sage.misc.parser), 264
forall() (in module sage.misc.misc), 130
format() (in module sage.misc.sagedoc), 377
format() (sage.misc.sage_input.SageInputFormatter method), 301
format_search_as_html() (in module sage.misc.sagedoc), 378
format_src() (in module sage.misc.sagedoc), 378
formatannotation() (in module sage.misc.sageinspect), 426
FourTi2 (class in sage.features.four_ti_2), 238
FourTi2Executable (class in sage.features.four_ti_2), 238
func_persist (class in sage.misc.func_persist), 337
function() (sage.ext.fast_callable.ExpressionCall method), 212
function_name() (in module sage.ext.fast_callable), 222

G

gammavariate() (in module sage.misc.prandom), 46
GapPackage (class in sage.features.gap), 238
gauss() (in module sage.misc.prandom), 46
gen() (in module sage.misc.functional), 9
gen() (sage.misc.sage_input.SageInputBuilder method), 294
gen_rest_table_index() (in module sage.misc.rest_index_of_methods), 447
gen_thematic_rest_table_index() (in module sage.misc.rest_index_of_methods), 448
generate_code() (in module sage.ext.fast_callable), 222
gens() (in module sage.misc.functional), 10
GET() (sage.misc.explain_pickle.PickleExplainer method), 352
get() (sage.misc.lazy_list.lazy_list_generic method), 169
get() (sage.misc.weak_dict.WeakValueDictionary method), 201
get_baseline() (sage.typeset.character_art.CharacterArt method), 272
get_breakpoints() (sage.typeset.character_art.CharacterArt method), 272
get_builtin_functions() (in module sage.ext.fast_callable), 223
get_cache_file() (in module sage.misc.lazy_import_cache), 163
get_connection() (sage.databases.sql_db.SQLDatabase method), 106
get_current() (sage.ext.fast_callable.InstructionStream method), 216
get_curs() (sage.databases.sql_db.SQLDatabase method), 107
get_key() (sage.misc.cachefunc.CachedFunction method), 179
get_main_globals() (in module sage.misc.misc), 130
get_metadata() (sage.ext.fast_callable.InstructionStream method), 217
get_name() (sage.misc.sage_input.SageInputFormatter method), 301
get_orig_args() (sage.ext.fast_callable.Wrapper method), 219
get_query_string() (sage.databases.sql_db.SQLQuery method), 112
get_remote_file() (in module sage.misc.remote_file), 93
get_skeleton() (sage.databases.sql_db.SQLDatabase method), 107
get_star_imports() (in module sage.misc.lazy_import), 160
get_systems() (in module sage.misc.citation), 394
get_verbose() (in module sage.misc.verbose), 147
get_verbose_files() (in module sage.misc.verbose), 147
getattr() (sage.misc.sage_input.SageInputBuilder method), 294
getframerate() (sage.media.wav.Wave method), 120
getframerate() (sage.media.wav.Wave method), 121
getframerate() (sage.media.wav.Wave method), 121
getframerate() (in module sage.misc.prandom), 46
getsampwidth() (sage.media.wav.Wave method), 121
GLOBAL() (sage.misc.explain_pickle.PickleExplainer method), 352
GlobalCputime (class in sage.misc.misc), 127
GloballyCachedMethodCaller (class in sage.misc.cachefunc), 190
graphics_from_save() (in module sage.structure.graphics_file), 119
GraphicsFile (class in sage.structure.graphics_file), 118
Graphviz (class in sage.features.graphviz), 240
Gurobi (class in sage.features.mip_backends), 247

H

has_file() (sage.misc.latex.Latex method), 319
has_instr() (sage.ext.fast_callable.InstructionStream method), 217
has_key() (sage.misc.converting_dict.KeyConvertingDict method), 73
has_latex_attr() (in module sage.misc.latex), 326
have_convert() (in module sage.misc.latex), 326
have_dvipng() (in module sage.misc.latex), 327
have_latex() (in module sage.misc.latex), 327
have_pdflatex() (in module sage.misc.latex), 327
have_program() (in module sage.misc.sage_ostools), 94
have_xelatex() (in module sage.misc.latex), 327
is_functional() (sage.features.lrs.Lrs method), 246
is_in_cache() (sage.misc.cachefunc.CachedFunction method), 180
is_in_cache() (sage.misc.cachefunc.CachedMethodCaller method), 186
is_in_string() (in module sage.misc.misc), 132
is_installed() (sage.misc.package.PackageInfo method), 387
is_integrally_closed() (in module sage.misc.functional), 13
is_iterator() (in module sage.misc.iterate), 170
is_lazy_string() (in module sage.misc.lazy_string), 170
is_mutable_pickle_object() (sage.misc.explain_pickle.PickleExplainer method), 353
is_odd() (in module sage.misc.functional), 13
is_optional() (sage.misc.abstract_method.AbstractMethod method), 53
is_package_installed() (in module sage.misc.package), 388
is_present() (sage.features.Feature method), 227
is_sublist() (in module sage.misc.sageinspect), 427
isqrt() (in module sage.misc.functional), 14
items() (sage.misc.cachefunc.FileCache method), 189
items() (sage.misc.weak_dict.WeakValueDictionary method), 201
iterator_prod() (in module sage.misc.misc), 141
iteritems() (sage.misc.weak_dict.WeakValueDictionary method), 202
itervalues() (sage.misc.weak_dict.WeakValueDictionary method), 203
J
JoinFeature (class in sage.features.join_feature), 231
JuPyMake (class in sage.features.polymake), 248
K
Kenzo (class in sage.features.kenzo), 243
kernel() (in module sage.misc.functional), 14
KeyConvertingDict (class in sage.misc.convert_dict), 73
keys() (sage.misc.cachefunc.FileCache method), 190
keys() (sage.misc.weak_dict.WeakValueDictionary method), 204
krull_dimension() (in module sage.misc.functional), 14
L
last() (sage.misc.parser.Tokenizer method), 262
last_token_string() (sage.misc.parser.Tokenizer method), 262
latex (class in sage.features.latex), 244
LaTeX (class in sage.misc.latex), 315
latex() (in module sage.misc.latex), 327
latex_extra_preamble() (in module sage.misc.latex), 328
latex_variable_name() (in module sage.misc.latex), 328
latex_variable_names() (in module sage.misc.latex), 329
latex_varify() (in module sage.misc.latex), 329
LatexCall (class in sage.misc.latex), 322
LatexExamples (class in sage.misc.latex), 322
LatexExamples.diagram (class in sage.misc.latex), 323
LatexExamples.graph (class in sage.misc.latex), 323
LatexExamples.knot (class in sage.misc.latex), 323
LatexExamples.pstricks (class in sage.misc.latex), 323
LatexExpr (class in sage.misc.latex), 323
LaTeXPackage (class in sage.features.latte), 243
Latte (class in sage.misc.latte), 245
Latte_count (class in sage.misc.latte), 245
Latte_integrate (class in sage.misc.latte), 245
launch_viewer() (sage.structure.graphics_file.GraphicsFile method), 118
lazy_attribute (class in sage.misc.lazy_attribute), 149
lazy_class_attribute (class in sage.misc.lazy_attribute), 157
lazy_import() (in module sage.misc.lazy_import), 161
lazy_list() (in module sage.misc.lazy_list), 165
lazy_list_formatter() (in module sage.misc.lazy_list), 166
lazy_list_from_function() (in module sage.misc.lazy_list), 167
lazy_list_from_iterator() (in module sage.misc.lazy_list), 168
lazy_list_from_update_function() (in module sage.misc.lazy_list), 168
lazy_list_generic() (class in sage.misc.lazy_list), 168
lazy_string() (in module sage.misc.lazy_string), 170
LazyFormat (class in sage.misc.lazy_format), 158
LazyImport (class in sage.misc.lazy_import), 160
LibFES (class in sage.features.fes), 237
LibFESLibrary (class in sage.features.fes), 237
License (class in sage.misc.copying), 395
lift() (in module sage.misc.functional), 15
LIST() (sage.misc.explain_pickle.PickleExplainer method), 352
list() (sage.misc.lazy_list.lazy_list_generic method), 169
list_function() (in module sage.misc.latex), 330
list_function() (in module sage.misc.mathml), 314
list_of_subfunctions() (in module sage.misc.rest_index_of_methods), 449
sage.features.lrs, 246
sage.features.mcqd, 246
sage.features.meataxe, 246
sage.features.mip_backends, 247
sage.features.normaliz, 247
sage.features.pandoc, 247
sage.features.pdf2svg, 248
sage.features.pkg_systems, 234
sage.features.polymake, 248
sage.features.rubiks, 248
sage.features.sagemath, 232
sage.features.tdlib, 250
sage.misc.abstract_method, 53
sage.misc.banner, 369
sage.misc.benchmark, 407
sage.misc.bindable_class, 56
sage.misc.c3, 451
sage.misc.c3_controlled, 453
sage.misc.cachefunc, 171
sage.misc.call, 71
sage.misc.callable_dict, 72
sage.misc.citation, 394
sage.misc.classcall_metaclass, 64
sage.misc.classgraph, 437
sage.misc.constant_function, 64
sage.misc.converting_dict, 73
sage.misc.copying, 395
sage.misc.cython, 251
sage.misc.decorators, 58
sage.misc.defaults, 1
sage.misc.dev_tools, 438
sage.misc.dist, 393
sage.misc.edit_module, 434
sage.misc.element_with_label, 87
sage.misc.explain_pickle, 349
sage.misc.fast_methods, 69
sage.misc.flatten, 75
sage.misc.fpickle, 348
sage.misc.func_persist, 337
sage.misc.function_mangling, 444
sage.misc.functional, 2
sage.misc.gperftools, 414
sage.misc.html, 310
sage.misc.inherit_comparison, 67
sage.misc.inline_fortran, 254
sage.misc.latex, 315
sage.misc.latex_macros, 334
sage.misc.lazy_attribute, 149
sage.misc.lazy_format, 158
sage.misc.lazy_import, 159
sage.misc.lazy_import_cache, 163
sage.misc.lazy_list, 164
sage.misc.lazy_string, 170
sage.misc.map_threaded, 83
sage.misc.mathml, 314
sage.misc.messaging, 93
sage.misc.method_decorator, 68
sage.misc.misc, 126
sage.misc.misc_c, 140
sage.misc.mrange, 77
sage.misc.multireplace, 82
sage.misc.nested_class, 361
sage.misc.nested_class_test, 407
sage.misc.object_multiplexer, 69
sage.misc.package, 386
sage.misc.pager, 376
sage.misc.parser, 255
sage.misc.persist, 337
sage.misc.prandom, 44
sage.misc.profiler, 413
sage.misc.python, 264
sage.misc.random_testing, 404
sage.misc.randstate, 33
sage.misc.remote_file, 93
sage.misc.repr, 283
sage.misc.reset, 371
sage.misc.rest_index_of_methods, 447
sage.misc.sage_eval, 265
sage.misc.sage_input, 285
sage.misc.sage_ostools, 94
sage.misc.sage_timeit, 410
sage.misc.sage_timeit_class, 412
sage.misc.sage_unittest, 397
sage.misc.sagedoc, 376
sage.misc.sageinspect, 419
sage.misc.search, 76
sage.misc.session, 365
sage.misc.sh, 268
sage.misc.sphinxify, 385
sage.misc.stopgap, 122
sage.misc.superseded, 123
sage.misc.table, 305
sage.misc.temporary_file, 88
sage.misc.unknown, 50
sage.misc.verbose, 146
sage.misc.viewer, 373
sage.misc.weak_dict, 197
sage.misc.graphics_file, 118
sage.typeset.ascii_art, 277
sage.typeset.character_art, 272
sage.typeset.character_art_factory, 274
sage.typeset.chars, 269
sage.typeset.ascii_art, 281
sage.typeset.unicode_characters, 282
mrange() (in module sage.misc.mrange), 78
mrange_iter() (in module sage.misc.mrange), 79
mro()  (sage.misc.c3_controlled.HierarchyElement method), 466
mro_controlled() (sage.misc.c3_controlled.HierarchyElement method), 466
mro_standard() (sage.misc.c3_controlled.HierarchyElement method), 467
multiple_replace() (in module sage.misc.multireplace), 82
Multiplex (class in sage.misc.object_multiplexer), 69
MultiplexFunction (class in sage.misc.object_multiplexer), 69
multiplicative_order() (multiplicative_order method), 58
my_getsource() (in module sage.misc.sagedoc), 379

N
N() (in module sage.misc.functional), 2
n() (in module sage.misc.functional), 18
name (sage.misc.package.PackageInfo attribute), 387
name() (sage.misc.sage_input.SageInputBuilder method), 296
name_is_valid() (in module sage.misc.explain_pickle), 359
neato (class in sage.features.graphviz), 240
nest() (in module sage.misc.sage), 133
nested_pickle() (in module sage.misc.nested_class), 363
NestedClassMetaclass (class in sage.misc.nested_class), 364
NEWFALSE() (sage.misc.explain_pickle.PickleExplainer method), 352
NEWOBJ() (sage.misc.explain_pickle.PickleExplainer method), 352
newton_method_sizes() (in module sage.misc.sage), 134
NEWTRUE() (sage.misc.explain_pickle.PickleExplainer method), 352
next() (sage.misc.parser.Tokenizer method), 262
ngens() (in module sage.misc.functional), 21
NonAssociative (class in sage.misc.misc), 140
NONE() (sage.misc.explain_pickle.PickleExplainer method), 352
None_function() (in module sage.misc.latex), 324
NonpicklingDict (class in sage.misc.cachefunc), 190
norm() (in module sage.misc.functional), 21
normalize_index() (in module sage.misc.misc), 141
normalvariate() (in module sage.misc.prandom), 47
numerator() (in module sage.misc.functional), 23
numerical_approx() (in module sage.misc.functional), 23

O
OBJ() (sage.misc.explain_pickle.PickleExplainer method), 352
objgen() (in module sage.misc.functional), 25
objgens() (in module sage.misc.functional), 26
objective (class in sage.features.interfaces), 242
op_list() (in module sage.ext.fast_callable), 223
op_list() (sage.ext.fast_callable.Wrapper method), 220
optimal (class in sage.features.rubiks), 249
optional_packages() (in module sage.misc.package), 389
options (class in sage.misc.decorators), 60
options() (sage.misc.table.table method), 309
order() (in module sage.misc.functional), 26
Outer (class in sage.misc.bindable_class), 58
Outer.Inner (class in sage.misc.bindable_class), 58

P
p_arg() (sage.misc.parser.Parser method), 256
p_args() (sage.misc.parser.Parser method), 256
p_atom() (sage.misc.parser.Parser method), 257
p_eqn() (sage.misc.parser.Parser method), 257
p_expr() (sage.misc.parser.Parser method), 257
p_factor() (sage.misc.parser.Parser method), 258
p_list() (sage.misc.parser.Parser method), 258
p_matrix() (sage.misc.parser.Parser method), 258
p_power() (sage.misc.parser.Parser method), 258
p_sequence() (sage.misc.parser.Parser method), 259
p_term() (sage.misc.parser.Parser method), 259
p_tuple() (sage.misc.parser.Parser method), 259
package_manifest() (in module sage.misc.package), 390
package_systems() (in module sage.features), 231
package_versions() (in module sage.misc.package), 390
PackageInfo (class in sage.misc.package), 387
PackageNotFoundError, 387
PackageSystem (class in sage.features.pkg_system), 234
pad_zeros() (in module sage.misc.sage), 134
pager() (in module sage.misc.pager), 376
Pandoc (class in sage.features.pandoc), 247
parent_with_gens() (sage.misc.sage_input.SageInputBuilder method), 296
paretotriang() (in module sage.misc.prandom), 47
parse() (sage.misc.parser.Parser method), 260
parse_expression() (sage.misc.parser.Parser method), 260
parse_keywords() (sage.typeset.character_art_factory.CharacterArtFactory method), 276
parse_sequence() (sage.misc.parser.Parser method), 260
Parser (class in sage.misc.parser), 255
PATH, 94
pdf2svg (class in sage.features.pdf2svg), 248
pdf_viewer() (in module sage.misc.viewer), 375
pdf_viewer() (sage.misc.viewer.Viewer method), 374
pdflatex (class in sage.features.latex), 245
peek() (sage.misc.parser.Tokenizer method), 262
PERSID() (sage.misc.explain_pickle.PickleExplainer method), 253
pickle_function() (in module sage.misc.pickle), 348
PickleDict (class in sage.misc.explain_pickle), 351
PickleExplainer (class in sage.misc.explain_pickle), 351
PickleInstance (class in sage.misc.explain_pickle), 356
picklejar() (in module sage.misc.persist), 343
pickleMethod() (in module sage.misc.fpickle), 348
pickleModule() (in module sage.misc.fpickle), 348
PickleObject (class in sage.misc.explain_pickle), 356
pip_installed_packages() (in module sage.package), 391
pip_remote_version() (in module sage.package), 391
PipPackageSystem (class in sage.features.pkg_systems), 234
pkgname_split() (in module sage.misc.package), 392
Plantri (class in sage.features.graph_generators), 239
plot() (sage.media.wav.Wave method), 121
plot_fft() (sage.media.wav.Wave method), 121
plot_raw() (sage.media.wav.Wave method), 121
png() (in module sage.misc.latex), 330
png_viewer() (in module sage.misc.viewer), 375
pretty_print_default() (in module sage.misc.html), 314
print_last() (sage.misc.profiler.Profiler method), 414
print_to_stdout() (sage.typeset.symbols.CompoundSymbol method), 271
process_dollars() (in module sage.misc.sagedoc), 379
process_extlinks() (in module sage.misc.sagedoc), 380
process_math() (in module sage.misc.sagedoc), 380
prod() (in module sage.misc.sage_c), 144
prod() (sage.misc.sage_input.SageInputBuilder method), 298
produce_latex_macros() (in module sage.misc.latex_macros), 334
Profiler (class in sage.misc.gperftools), 415
Profiler (class in sage.misc.profiler), 413
PROTO() (sage.misc.explain_pickle.PickleExplainer method), 353
push() (sage.misc.explain_pickle.PickleExplainer method), 354
push_and_share() (sage.misc.explain_pickle.PickleExplainer method), 354
push_mark() (sage.misc.explain_pickle.PickleExplainer method), 355
pushover() (in module sage.misc.messaging), 93
PUT() (sage.misc.explain_pickle.PickleExplainer method), 353
PyNormaliz (class in sage.features.normaliz), 247
Python (class in sage.misc.python), 264
python_calls() (sage.ext.fast_callable.Wrapper method), 404
python_random() (sage.misc.randstate.randstate method), 42
PythonModule (class in sage.features), 229
PythonObjectWithTests (class in sage.misc.sage_unittest), 400
Q
query() (sage.databases.sql_db.SQLDatabase method), 109
query_results() (sage.databases.sql_db.SQLQuery method), 113
quo() (in module sage.misc.functional), 26
quotient() (in module sage.misc.functional), 26
R
randint() (in module sage.misc.prandom), 47
random() (in module sage.misc.prandom), 47
random() (in module sage.misc.randstate), 40
random_sublist() (in module sage.misc.mrange), 135
random_testing() (class in module sage.misc.random_testing), 404
randrange() (in module sage.misc.prandom), 47
randstate (class in sage.misc.randstate), 41
rank() (in module sage.misc.functional), 27
readframes() (sage.media.wav.Wave method), 121
redirection (class in sage.misc.sage_ostools), 94
REDUCE() (sage.misc.explain_pickle.PickleExplainer
method), 353
reduce_code() (in module sage.misc.fpickle), 349
regexp() (in module sage.databases.sql_db), 116
register_name() (sage.misc.sage_input.SageInputFormatter
method), 302
register_unpickle_override() (in module
sage.misc.persist), 343
regulator() (in module sage.misc.functional), 27
remote_version (sage.misc.package.PackageInfo
attribute), 387
rename_keyword (class in sage.misc.decorators), 60
rename_table() (sage.databases.sql_db.SQLDatabase
method), 110
repr_lincomb() (in module sage.misc.latex), 330
require() (sage.features.Feature method), 284
require_version() (in module sage.misc.banner), 369
reset() (in module sage.misc.sage_reset), 371
reset() (sage.misc.parser.Tokenizer method), 263
reset_interfaces() (in module sage.misc.sage_reset), 372
resolution (sage.features.FeatureNotPresentError
attribute), 229
resolution (sage.features.FeatureTestResult
attribute), 229
require() (sage.features.Feature method), 228
restore() (in module sage.misc.sage_reset), 372
restore_cwd() (in module sage.misc.sage_ostools), 96
result() (sage.misc.sage_input.SageInputBuilder
method), 299
round() (in module sage.misc.functional), 27
Rubiks (class in sage.features.rubiks), 248
run() (sage.misc.sage_unittest.TestSuite method), 402
run_100ms() (in module sage.misc.gperftools), 416
run_pickle() (sage.misc.explain_pickle.PickleExplainer
method), 355
running_total() (in module sage.misc.misc_c), 144
runsnake() (in module sage.misc.dev_tools), 443
runTest() (sage.misc.sage_unittest.InstanceTester
method), 397

S
sage.arith.srange
module, 83
sage.databases.sql_db
module, 96
sage.docs.instancedoc
module, 417
sage.ext.fast_callable
module, 206
sage.ext.fast_eval
module, 224
sage.ext.memory
module, 451

sage.features
module, 225
sage.features.all
module, 232
sage.features.bliss
module, 235
sage.features.csdp
module, 235
sage.features.databases
module, 236
sage.features.dvipng
module, 237
sage.features.fes
module, 237
sage.features.ffmpeg
module, 237
sage.features.four_ti_2
module, 238
sage.features.gap
module, 238
sage.features.graph_generators
module, 238
sage.features.graphviz
module, 240
sage.features.imagemagick
module, 240
sage.features.interfaces
module, 241
sage.features.internet
module, 243
sage.features.join_feature
module, 231
sage.features.kenzo
module, 243
sage.features.latex
module, 243
sage.features.latte
module, 245
sage.features.lrs
module, 246
sage.features.mctd
module, 246
sage.features.mctd
module, 246
sage.features.mctd
module, 246
sage.features.mctd
module, 246
sage.features.mctd
module, 246
sage.features.normaliz
module, 247
sage.features.pandoc
module, 247
sage.features.pdf2svg
module, 248
sage.features.pkg_systems
module, 234

Index 485
sage.features.polymake
module, 248
sage.features.rubiks
module, 248
sage.features.sagemath
module, 232
sage.features.tdlib
module, 250
sage.media.wav
module, 119
sage.misc.abstract_method
module, 53
sage.misc.banner
module, 369
sage.misc.benchmark
module, 407
sage.misc.bindable_class
module, 56
sage.misc.c3
module, 451
sage.misc.c3_controlled
module, 453
sage.misc.cachefunc
module, 171
sage.misc.call
module, 71
sage.misc.callable_dict
module, 72
sage.misc.citation
module, 394
sage.misc.classcall_metaclass
module, 64
sage.misc.classgraph
module, 437
sage.misc.constant_function
module, 64
sage.misc.converting_dict
module, 73
sage.misc.copying
module, 395
sage.misc.cython
module, 251
sage.misc.decorators
module, 58
sage.misc.defaults
module, 1
sage.misc.dev_tools
module, 438
sage.misc.dist
module, 393
sage.misc.edit_module
module, 434
sage.misc.element_with_label
module, 87
sage.misc.explain_pickle
module, 349
sage.misc.fast_methods
module, 69
sage.misc.flatten
module, 75
sage.misc.fpickle
module, 348
sage.misc.func_persist
module, 337
sage.misc.function_mangling
module, 444
sage.misc.functional
module, 2
sage.misc.gperftools
module, 414
sage.misc.html
module, 310
sage.misc.inherit_comparison
module, 67
sage.misc.inline_fortran
module, 254
sage.misc.latex
module, 315
sage.misc.latex_macros
module, 334
sage.misc.lazy_attribute
module, 149
sage.misc.lazy_format
module, 158
sage.misc.lazy_import
module, 159
sage.misc.lazy_import_cache
module, 163
sage.misc.lazy_list
module, 164
sage.misc.lazy_string
module, 170
sage.misc.map_threaded
module, 83
sage.misc.mathml
module, 314
sage.misc.messaging
module, 93
sage.misc.method_decorator
module, 68
sage.misc.misc
module, 126
sage.misc.misc_c
module, 140
sage.misc.mrange
module, 77
sage.misc.multireplace
module, 82
TUPLE() (sage.misc.explain_pickle.PickleExplainer method), 353
TUPLE1() (sage.misc.explain_pickle.PickleExplainer method), 353
TUPLE2() (sage.misc.explain_pickle.PickleExplainer method), 353
TUPLE3() (sage.misc.explain_pickle.PickleExplainer method), 353
tuple_function() (in module sage.misc.latex), 332
tuple_function() (in module sage.misc.mathml), 314
twopi (class in sage.features.graphviz), 240
type (sage.misc.package.PackageInfo attribute), 387
typecall() (in module sage.misc.classcall_metaclass), 65

update() (sage.misc.converting_dict.KeyConvertingDict method), 74
use_variable() (sage.misc.sage_input.SageInputBuilder method), 300

V
vacuum() (sage.databases.sql_db.SQLDatabase method), 111
validate() (sage.structure.graphics_file.Mime class method), 118
value() (sage.ext.fast_callable.ExpressionConstant method), 213
values() (sage.media.wav.Wave method), 122
values() (sage.misc.cachefunc.FileCache method), 190
values() (sage.misc.sage_input.SageInputBuilder method), 148

V
variable_index() (sage.ext.fast_callable.ExpressionVariable dictionary), 216
variable_names() (in module sage.misc.defaults), 2
vector() (sage.media.wav.Wave method), 122
vector_delimiters() (sage.misc.latex.Latex method), 321

V
verbose() (in module sage.misc.verbose), 148
verify_column() (in module sage.databases.sql_db), 116
verify_operator() (in module sage.databases.sql_db), 117
verify_same() (in module sage.misc.sage_input), 303
verify_si_answer() (in module sage.misc.sage_input), 304
verify_type() (in module sage.databases.sql_db), 117
version() (in module sage.misc.banner), 370
version_dict() (in module sage.misc.banner), 370
view() (in module sage.misc.latex), 332
Viewer (class in sage.misc.viewer), 373
visit_arg() (sage.misc.sageinspect.SageArgSpecVisitor method), 425
visit_BinOp() (sage.misc.sageinspect.SageArgSpecVisitor method), 422
visit_BoolOp() (sage.misc.sageinspect.SageArgSpecVisitor method), 422
visit_Compare() (sage.misc.sageinspect.SageArgSpecVisitor method), 422
visit_Dict() (sage.misc.sageinspect.SageArgSpecVisitor method), 423
visit_List() (sage.misc.sageinspect.SageArgSpecVisitor method), 423
visit_Name() (sage.misc.sageinspect.SageArgSpecVisitor method), 423
visit_NameConstant() (sage.misc.sageinspect.SageArgSpecVisitor method), 424

490 Index
visit_Num() (sage.misc.sageinspect.SageArgSpecVisitor method), 424
visit_Str() (sage.misc.sageinspect.SageArgSpecVisitor method), 424
visit_Tuple() (sage.misc.sageinspect.SageArgSpecVisitor method), 425
visit_UnaryOp() (sage.misc.sageinspect.SageArgSpecVisitor method), 425
vonmisesvariate() (in module sage.misc.prandom), 49

W
walltime() (in module sage.misc.misc), 139
warning() (in module sage.misc.superseded), 126
Wave (class in sage.media.wav), 120
weak_cached_function() (in module sage.misc.cachefunc), 196
WeakCachedFunction (class in sage.misc.cachefunc), 191
WeakValueDictEraser (class in sage.misc.weak_dict), 200
WeakValueDictionary (class in sage.misc.weak_dict), 200
weibullvariate() (in module sage.misc.prandom), 49
width() (sage.typeset.character_art.CharacterArt method), 273
WithEqualityById (class in sage.misc.fast_methods), 70
word_wrap() (in module sage.misc.misc), 139
Wrapper (class in sage.ext.fast_callable), 219

X
xelatex (class in sage.features.latex), 245
xinterval() (in module sage.misc.functional), 33
xmrange (class in sage.misc.mrange), 79
xmrange_iter (class in sage.misc.mrange), 81
xsrange() (in module sage.arith.srange), 86

Z
ZZ_seed() (sage.misc.randstate.randstate method), 41