
Interpreter Interfaces

Release 9.7

The Sage Development Team

Jul 21, 2024

CONTENTS

1	Common Interface Functionality	3
2	Common Interface Functionality through Pexpect	9
3	Sage wrapper around pexpect's spawn class and	13
4	Interface to Axiom	15
5	The Elliptic Curve Factorization Method	21
6	Interface to 4ti2	27
7	Interface to FriCAS	33
8	Interface to Frobbly for fast computations on monomial ideals.	45
9	Interface to GAP	49
9.1	First Examples	49
9.2	GAP and Singular	49
9.3	Saving and loading objects	51
9.4	Long Input	51
9.5	Changing which GAP is used, and how	51
10	Interface to GAP3	59
10.1	Obtaining GAP3	59
10.2	Changing which GAP3 is used	59
10.3	Functionality and Examples	60
10.4	Common Pitfalls	61
10.5	Examples	62
11	Interface to Groebner Fan	67
12	Pexpect Interface to Giac	69
12.1	Tutorial	70
13	Interface to the Gnuplot interpreter	79
14	Interface to the GP calculator of PARI/GP	81
15	Interface for extracting data and generating images from Jmol readable files.	91

16	Interface to KASH	93
16.1	Issues	93
16.2	Tutorial	93
16.3	Long Input	99
17	Library interface to Kenzo	103
18	Interface to LattE integrale programs	131
19	Interface to LiE	135
19.1	Tutorial	136
20	Lisp Interface	143
21	Interface to Macaulay2	147
22	Interface to Magma	161
22.1	Parameters	161
22.2	Multiple Return Values	162
22.3	Long Input	162
22.4	Garbage Collection	162
22.5	Other Examples	163
23	Interface to the free online MAGMA calculator	181
24	Interface to Maple	183
24.1	Tutorial	183
25	Interface to Mathematica	191
25.1	Tutorial	192
25.2	Long Input	195
25.3	Loading and saving	195
25.4	Complicated translations	195
26	Interface to Mathics	201
26.1	Tutorial	202
26.2	Long Input	205
26.3	Loading and saving	205
26.4	Complicated translations	205
27	Interface to MATLAB	213
27.1	Tutorial	213
28	Pexpect interface to Maxima	219
28.1	Tutorial	220
28.2	Examples involving matrices	222
28.3	Laplace Transforms	223
28.4	Continued Fractions	224
28.5	Special examples	224
28.6	Miscellaneous	225
28.7	Interactivity	225
28.8	Latex Output	225
28.9	Long Input	225
29	Abstract interface to Maxima	229

30 Library interface to Maxima	247
31 Interface to MuPAD	257
32 Interface to mwrnk	261
33 Interface to GNU Octave	265
33.1 Computation of Special Functions	265
33.2 Tutorial	266
34 Interface to PHC.	271
35 Interface to polymake	279
36 POV-Ray, The Persistence of Vision Ray Tracer	291
37 Parallel Interface to the Sage interpreter	293
38 Interface to QEPCAD	295
39 Interface to Bill Hart’s Quadratic Sieve	321
40 Interfaces to R	325
41 Interface to several Rubik’s cube solvers.	339
42 Interface to Sage	343
43 Interface to Scilab	349
44 Interface to Singular	355
44.1 Introduction	355
44.2 Tutorial	355
44.3 Computing the Genus	358
44.4 An Important Concept	358
44.5 Long Input	359
45 SymPy → Sage conversion	377
46 The Tachyon Ray Tracer	379
46.1 Basic Scene Requirements	379
46.2 Camera and viewing parameters	380
46.3 Including Files	381
46.4 Scene File Comments	382
46.5 Lights	382
46.6 Atmospheric effects	383
46.7 Objects	383
46.8 Texture and Color	387
47 Interface to TIDES	391
48 Interface to the Sage cleaner	395
49 Quitting interfaces	397
50 An interface to read data files	399

51 Indices and Tables	401
Python Module Index	403
Index	405

Sage provides a unified interface to the best computational software. This is accomplished using both C-libraries (see [C/C++ Library Interfaces](#)) and interpreter interfaces, which are implemented using pseudo-tty's, system files, etc. This chapter is about these interpreter interfaces.

Note: Each interface requires that the corresponding software is installed on your computer. Sage includes GAP, PARI, Singular, and Maxima, but does not include Octave (very easy to install), MAGMA (non-free), Maple (non-free), or Mathematica (non-free).

There is overhead associated with each call to one of these systems. For example, computing $2+2$ thousands of times using the GAP interface will be slower than doing it directly in Sage. In contrast, the C-library interfaces of [C/C++ Library Interfaces](#) incur less overhead.

In addition to the commands described for each of the interfaces below, you can also type e.g., `%gap`, `%magma`, etc., to directly interact with a given interface in its state. Alternatively, if `X` is an interface object, typing `X.interact()` allows you to interact with it. This is completely different than `X.console()` which starts a complete new copy of whatever program `X` interacts with. Note that the input for `X.interact()` is handled by Sage, so the history buffer is the same as for Sage, tab completion is as for Sage (unfortunately!), and input that spans multiple lines must be indicated using a backslash at the end of each line. You can pull data into an interactive session with `X` using `sage(expression)`.

The console and interact methods of an interface do very different things. For example, using `gap` as an example:

1. `gap.console()`: You are completely using another program, e.g., `gap/magma/gp`. Here Sage is serving as nothing more than a convenient program launcher, similar to `bash`.
2. `gap.interact()`: This is a convenient way to interact with a running `gap` instance that may be “full of” Sage objects. You can import Sage objects into this `gap` (even from the interactive interface), etc.

The console function is very useful on occasion, since you get the exact actual program available (especially useful for tab completion and testing to make sure nothing funny is going on).

COMMON INTERFACE FUNCTIONALITY

See the examples in the other sections for how to use specific interfaces. The interface classes all derive from the generic interface that is described in this section.

AUTHORS:

- William Stein (2005): initial version
- William Stein (2006-03-01): got rid of infinite loop on startup if client system missing
- Felix Lawrence (2009-08-21): edited `._sage_()` to support lists and float exponents in foreign notation.
- Simon King (2010-09-25): `Expect._local_tmpfile()` depends on `Expect.pid()` and is cached; `Expect.quit()` clears that cache, which is important for forking.
- Jean-Pierre Flori (2010,2011): Split non Pexpect stuff into a parent class.
- Simon King (2015): Improve pickling for `InterfaceElement`

```
class sage.interfaces.interface.AsciiArtString
```

```
    Bases: str
```

```
class sage.interfaces.interface.Interface(name)
```

```
    Bases: sage.misc.fast_methods.WithEqualityById, sage.structure.parent_base.ParentWithBase
```

```
    Interface interface object.
```

Note: Two interfaces compare equal if and only if they are identical objects (this is a critical constraint so that caching of representations of objects in interfaces works correctly). Otherwise they are never equal.

```
call(function_name, *args, **kwds)
```

```
clear(var)
```

```
    Clear the variable named var.
```

```
console()
```

```
cputime()
```

```
    CPU time since this process started running.
```

```
eval(code, **kwds)
```

```
    Evaluate code in an interface.
```

```
    This method needs to be implemented in sub-classes.
```

```
    Note that it is not always to be expected that it returns a non-empty string. In contrast, get() is supposed to return the result of applying a print command to the object so that the output is easier to parse.
```



```

sage: from sage.interfaces.interface import Interface
sage: i = Interface("")
sage: i.rand_seed() # random
318491487

sage: s = Singular()
sage: s.rand_seed() # random
365260051

```

read(*filename*)

EXAMPLES:

```

sage: filename = tmp_filename()
sage: f = open(filename, 'w')
sage: _ = f.write('x = 2\n')
sage: f.close()
sage: octave.read(filename) # optional - octave
sage: octave.get('x')       # optional - octave
' 2'

sage: import os
sage: os.unlink(filename)

```

set(*var, value*)

Set the variable *var* to the given value.

set_seed(*seed=None*)

Set the random seed for the interpreter and return the new value of the seed.

This is dependent on which interpreter so must be implemented in each separately. For examples see `gap.py` or `singular.py`.

If *seed* is `None` then should generate a random seed.

EXAMPLES:

```

sage: s = Singular()
sage: s.set_seed(1)
1
sage: [s.random(1,10) for i in range(5)]
[8, 10, 4, 9, 1]

sage: from sage.interfaces.interface import Interface
sage: i = Interface("")
sage: i.set_seed()
Traceback (most recent call last):
...
NotImplementedError: This interpreter did not implement a set_seed function

```

class `sage.interfaces.interface.InterfaceElement`(*parent, value, is_name=False, name=None*)

Bases: `sage.structure.element.Element`

Interface element.

attribute(*attrname*)

If this wraps the object *x* in the system, this returns the object *x.attrname*. This is useful for some systems that have object oriented attribute access notation.

EXAMPLES:

```

sage: x = r([1,2,3]); x #_
↳optional - rpy2
[1] 1 2 3
sage: x.name() #_
↳optional - rpy2
'sage...'
sage: x = r([1,2,3]).name('x'); x #_
↳optional - rpy2
[1] 1 2 3
sage: x.name() #_
↳optional - rpy2
'x'

```

```

sage: s5 = gap.SymmetricGroup(5).name('s5')
sage: s5
SymmetricGroup( [ 1 .. 5 ] )
sage: s5.name()
's5'

```

sage(*args, **kws)

Attempt to return a Sage version of this object.

This method does nothing more than calling `_sage_()`, simply forwarding any additional arguments.

EXAMPLES:

```

sage: gp(1/2).sage()
1/2
sage: _.parent()
Rational Field
sage: singular.lib("matrix")
sage: R = singular.ring(0, '(x,y,z)', 'dp')
sage: singular.matrix(2,2).sage()
[0 0]
[0 0]

```

class sage.interfaces.interface.**InterfaceFunction**(parent, name)

Bases: sage.structure.sage_object.SageObject

Interface function.

class sage.interfaces.interface.**InterfaceFunctionElement**(obj, name)

Bases: sage.structure.sage_object.SageObject

Interface function element.

help()

sage.interfaces.interface.**is_InterfaceElement**(x)

COMMON INTERFACE FUNCTIONALITY THROUGH PEXPECT

See the examples in the other sections for how to use specific interfaces. The interface classes all derive from the generic interface that is described in this section.

AUTHORS:

- William Stein (2005): initial version
- William Stein (2006-03-01): got rid of infinite loop on startup if client system missing
- Felix Lawrence (2009-08-21): edited `._sage_()` to support lists and float exponents in foreign notation.
- Simon King (2010-09-25): `Expect._local_tmpfile()` depends on `Expect.pid()` and is cached; `Expect.quit()` clears that cache, which is important for forking.
- Jean-Pierre Flori (2010,2011): Split non Pexpect stuff into a parent class.
- Simon King (2010-11-23): Ensure that the interface is started again after a crash, when a command is executed in `_eval_line`. Allow synchronisation of the GAP interface.
- François Bissey, Bill Page, Jeroen Demeyer (2015-12-09): Upgrade to pexpect 4.0.1 + patches, see [trac ticket #10295](#).

```
class sage.interfaces.expect.Expect(name, prompt, command=None, env={}, server=None,
                                   server_tmpdir=None, ulimit=None, maxread=None,
                                   script_subdirectory=None, restart_on_ctrlc=False,
                                   verbose_start=False, init_code=[], max_startup_time=None,
                                   logfile=None, eval_using_file_cutoff=0, do_cleaner=True,
                                   remote_cleaner=False, path=None, terminal_echo=True)
```

Bases: `sage.interfaces.interface.Interface`

Expect interface object.

clear_prompts()

command()

Returns the command used in this interface.

EXAMPLES:

```
sage: magma.set_server_and_command(command = 'magma-2.19')
sage: magma.command() # indirect doctest
'magma-2.19'
```

detach()

Forget the running subprocess: keep it running but pretend that it's no longer running.

EXAMPLES:

```

sage: a = maxima('y')
sage: saved_expect = maxima._expect # Save this to close later
sage: maxima.detach()
sage: a._check_valid()
Traceback (most recent call last):
...
ValueError: The maxima session in which this object was defined is no longer
↳ running.
sage: saved_expect.close() # Close child process

```

Calling `detach()` a second time does nothing:

```

sage: maxima.detach()

```

`eval`(*code*, *strip=True*, *synchronize=False*, *locals=None*, *allow_use_file=True*, *split_lines='nofile'*, ***kwds*)
INPUT:

- *code* – text to evaluate
- **strip** – bool; whether to strip output prompts, etc. (ignored in the base class).
- **locals** – None (ignored); this is used for compatibility with the Sage notebook’s generic system interface.
- **allow_use_file** – bool (default: True); if True and code exceeds an interface-specific threshold then code will be communicated via a temporary file rather than the character-based interface. If False then the code will be communicated via the character interface.
- **split_lines** – Tri-state (default: “nofile”); if “nofile” then code is sent line by line unless it gets communicated via a temporary file. If True then code is sent line by line, but some lines individually might be sent via temporary file. Depending on the interface, this may transform grammatical code into ungrammatical input. If False, then the whole block of code is evaluated all at once.
- ****kwds** – All other arguments are passed onto the `_eval_line` method. An often useful example is `reformat=False`.

`expect()`

`interrupt`(*tries=5*, *timeout=2.0*, *quit_on_fail=True*)

`is_local()`

`is_remote()`

`is_running()`

Return True if self is currently running.

`path()`

`pid()`

Return the PID of the underlying sub-process.

REMARK:

If the interface terminates unexpectedly, the original PID will still be used. But if it was terminated using `quit()`, a new sub-process with a new PID is automatically started.

EXAMPLES:

```

sage: pid = gap.pid()
sage: gap.eval('quit;')
''
sage: pid == gap.pid()
True
sage: gap.quit()
sage: pid == gap.pid()
False

```

quit(*verbose=False*)

Quit the running subprocess.

INPUT:

- *verbose* – (boolean, default `False`) print a message when quitting this process?

EXAMPLES:

```

sage: a = maxima('y')
sage: maxima.quit(verbose=True)
Exiting Maxima with PID ... running .../bin/maxima...
sage: a._check_valid()
Traceback (most recent call last):
...
ValueError: The maxima session in which this object was defined is no longer
↳running.

```

Calling `quit()` a second time does nothing:

```

sage: maxima.quit(verbose=True)

```

server()

Returns the server used in this interface.

EXAMPLES:

```

sage: magma.set_server_and_command(server = 'remote')
No remote temporary directory (option server_tmpdir) specified, using /tmp/ on
↳remote
sage: magma.server() # indirect doctest
'remote'

```

set_server_and_command(*server=None, command=None, server_tmpdir=None, ulimit=None*)

Changes the server and the command to use for this interface. This raises a Runtime error if the interface is already started.

EXAMPLES:

```

sage: magma.set_server_and_command(server = 'remote', command = 'mymagma') #
↳indirect doctest
No remote temporary directory (option server_tmpdir) specified, using /tmp/ on
↳remote
sage: magma.server()
'remote'
sage: magma.command()
'ssh -t remote 'mymagma''

```

`user_dir()`

class `sage.interfaces.expect.ExpectElement`(*parent, value, is_name=False, name=None*)
 Bases: `sage.interfaces.interface.InterfaceElement`

Expect element.

class `sage.interfaces.expect.ExpectFunction`(*parent, name*)
 Bases: `sage.interfaces.interface.InterfaceFunction`

Expect function.

class `sage.interfaces.expect.FunctionElement`(*obj, name*)
 Bases: `sage.interfaces.interface.InterfaceFunctionElement`

Expect function element.

class `sage.interfaces.expect.StdoutContext`(*interface, silent=False, stdout=None*)
 Bases: `object`

A context in which all communication between Sage and a subprocess interfaced via pexpect is printed to stdout.

`sage.interfaces.expect.console`(*cmd*)

class `sage.interfaces.expect.gc_disabled`
 Bases: `object`

This is a “with” statement context manager. Garbage collection is disabled within its scope. Nested usage is properly handled.

EXAMPLES:

```
sage: import gc
sage: from sage.interfaces.expect import gc_disabled
sage: gc.isenabled()
True
sage: with gc_disabled():
....:     print(gc.isenabled())
....:     with gc_disabled():
....:         print(gc.isenabled())
....:     print(gc.isenabled())
False
False
False
sage: gc.isenabled()
True
```

`sage.interfaces.expect.is_ExpectElement`(*x*)

SAGE WRAPPER AROUND PEXPECT'S SPAWN CLASS AND

the `ptyprocess`'s `PtyProcess` class.

AUTHOR:

- Jeroen Demeyer (2015-02-01): initial version, see [trac ticket #17686](#).
- Jeroen Demeyer (2015-12-04): add support for `pexpect 4` + `ptyprocess`, see [trac ticket #10295](#).

class `sage.interfaces.sagespawn.SagePtyProcess`(*pid, fd*)

Bases: `ptyprocess.ptyprocess.PtyProcess`

close(*force=None*)

Quit the child process: send the quit string, close the pseudo-tty and kill the process.

This function returns immediately, it doesn't wait for the child process to die.

EXAMPLES:

```
sage: from sage.interfaces.sagespawn import SageSpawn
sage: s = SageSpawn("sleep 1000")
sage: s.close()
sage: while s.isalive(): # long time (5 seconds)
.....:     sleep(float(0.1))
```

terminate_async(*interval=5.0*)

Terminate the child process group asynchronously.

This function returns immediately, while the child is slowly being killed in the background.

INPUT:

- `interval` – (default: 5) how much seconds to wait between sending two signals.

EXAMPLES:

Run an infinite loop in the shell:

```
sage: from sage.interfaces.sagespawn import SageSpawn
sage: s = SageSpawn("sh", ["-c", "while true; do sleep 1; done"])
```

Check that the process eventually dies after calling `terminate_async`:

```
sage: s.ptyproc.terminate_async(interval=float(0.2))
sage: while True:
.....:     try:
.....:         os.kill(s.pid, 0)
.....:     except OSError:
```

(continues on next page)

(continued from previous page)

```

.....:         sleep(float(0.1))
.....:     else:
.....:         break # process got killed

```

class `sage.interfaces.sagespawn.SageSpawn(*args, **kws)`

Bases: `pexpect.pty_spawn.spawn`

Spawn a subprocess in a pseudo-tty.

- `*args, **kws`: see `pexpect.spawn`.
- `name` – human-readable name for this process, used for display purposes only.
- `quit_string` – (default: `None`) if not `None`, send this string to the child process before killing it.

EXAMPLES:

```

sage: from sage.interfaces.sagespawn import SageSpawn
sage: SageSpawn("sleep 1", name="Sleeping Beauty")
Sleeping Beauty with PID ... running ...

```

expect_peek(*args, **kws)

Like `expect()` but restore the read buffer such that it looks like nothing was actually read. The next reading will continue at the current position.

EXAMPLES:

```

sage: from sage.interfaces.sagespawn import SageSpawn
sage: E = SageSpawn("sh", ["-c", "echo hello world"])
sage: _ = E.expect_peek("w")
sage: E.read().decode('ascii')
'hello world\r\n'

```

expect_upto(*args, **kws)

Like `expect()` but restore the read buffer starting from the matched string. The next reading will continue starting with the matched string.

EXAMPLES:

```

sage: from sage.interfaces.sagespawn import SageSpawn
sage: E = SageSpawn("sh", ["-c", "echo hello world"])
sage: _ = E.expect_upto("w")
sage: E.read().decode('ascii')
'world\r\n'

```

INTERFACE TO AXIOM

Todo:

- Evaluation using a file is not done. Any input line with more than a few thousand characters would hang the system, so currently it automatically raises an exception.
 - All completions of a given command.
 - Interactive help.
-

Axiom is a free GPL-compatible (modified BSD license) general purpose computer algebra system whose development started in 1973 at IBM. It contains symbolic manipulation algorithms, as well as implementations of special functions, including elliptic functions and generalized hypergeometric functions. Moreover, Axiom has implementations of many functions relating to the invariant theory of the symmetric group S_n . For many links to Axiom documentation see <http://wiki.axiom-developer.org>.

AUTHORS:

- Bill Page (2006-10): Created this (based on Maxima interface)

Note: Bill Page put a huge amount of effort into the Sage Axiom interface over several days during the Sage Days 2 coding sprint. This contribution is greatly appreciated.

- William Stein (2006-10): misc touchup.
- Bill Page (2007-08): Minor modifications to support axiom4sage-0.3

Note: The axiom4sage-0.3.spkg is based on an experimental version of the FriCAS fork of the Axiom project by Waldek Hebisch that uses pre-compiled cached Lisp code to build Axiom very quickly with clisp.

If the string “error” (case insensitive) occurs in the output of anything from axiom, a RuntimeError exception is raised.

EXAMPLES: We evaluate a very simple expression in axiom.

```
sage: axiom('3 * 5')          #optional - axiom
15
sage: a = axiom(3) * axiom(5); a  #optional - axiom
15
```

The type of a is AxiomElement, i.e., an element of the axiom interpreter.

```
sage: type(a) #optional - axiom
<class 'sage.interfaces.axiom.AxiomElement'>
sage: parent(a) #optional - axiom
Axiom
```

The underlying Axiom type of `a` is also available, via the `type` method:

```
sage: a.type() #optional - axiom
PositiveInteger
```

We factor $x^5 - y^5$ in Axiom in several different ways. The first way yields a Axiom object.

```
sage: F = axiom.factor('x^5 - y^5'); F #optional - axiom
      4      3      2 2      3      4
- (y - x)(y + x y + x y + x y + x )
sage: type(F) #optional - axiom
<class 'sage.interfaces.axiom.AxiomElement'>
sage: F.type() #optional - axiom
Factored Polynomial Integer
```

Note that Axiom objects are normally displayed using “ASCII art”.

```
sage: a = axiom(2/3); a #optional - axiom
  2
 -
  3
sage: a = axiom('x^2 + 3/7'); a #optional - axiom
  2 3
 x + -
  7
```

The `axiom.eval` command evaluates an expression in axiom and returns the result as a string. This is exact as if we typed in the given line of code to axiom; the return value is what Axiom would print out.

```
sage: print(axiom.eval('factor(x^5 - y^5)')) # optional - axiom
      4      3      2 2      3      4
- (y - x)(y + x y + x y + x y + x )
Type: Factored Polynomial Integer
```

We can create the polynomial f as a Axiom polynomial, then call the `factor` method on it. Notice that the notation `f.factor()` is consistent with how the rest of Sage works.

```
sage: f = axiom('x^5 - y^5') #optional - axiom
sage: f^2 #optional - axiom
      10      5 5      10
 y - 2x y + x
sage: f.factor() #optional - axiom
      4      3      2 2      3      4
- (y - x)(y + x y + x y + x y + x )
```

Control-C interruption works well with the axiom interface, because of the excellent implementation of axiom. For example, try the following sum but with a much bigger range, and hit control-C.

(continued from previous page)

```

x + 1
sage: a.unparsed_input_form() #optional - axiom
'x*x+1'

```

class sage.interfaces.axiom.PanAxiomExpectFunction(*parent, name*)

Bases: *sage.interfaces.expect.ExpectFunction*

class sage.interfaces.axiom.PanAxiomFunctionElement(*object, name*)

Bases: *sage.interfaces.expect.FunctionElement*

sage.interfaces.axiom.**axiom_console**()

Spawn a new Axiom command-line session.

EXAMPLES:

```

sage: axiom_console() #not tested
          AXIOM Computer Algebra System
          Version: Axiom (January 2009)
          Timestamp: Sunday January 25, 2009 at 07:08:54
-----
Issue )copyright to view copyright notices.
Issue )summary for a summary of useful system commands.
Issue )quit to leave AXIOM and return to shell.
-----

```

sage.interfaces.axiom.**is_AxiomElement**(*x*)

Returns True if *x* is of type AxiomElement.

EXAMPLES:

```

sage: from sage.interfaces.axiom import is_AxiomElement
sage: is_AxiomElement(axiom(2)) #optional - axiom
True
sage: is_AxiomElement(2)
False

```

sage.interfaces.axiom.**reduce_load_Axiom**()

Returns the Axiom interface object defined in sage.interfaces.axiom.

EXAMPLES:

```

sage: from sage.interfaces.axiom import reduce_load_Axiom
sage: reduce_load_Axiom()
Axiom

```



```

sage: n = next_prime(11^23)*next_prime(11^37)
sage: ecm.time(n, 35) # random output
Expected curves: 910, Expected time: 23.95m

sage: ecm.time(n, 30, verbose=True) # random output
GMP-ECM 6.4.4 [configured with MPIR 2.6.0, --enable-asm-redc] [ECM]
Running on localhost.localdomain
Input number is 304481639541418099574459496544854621998616257489887231115912293_
↪(63 digits)
Using MODMULN [mulredc:0, sqrrcdc:0]
Using B1=250000, B2=128992510, polynomial Dickson(3), sigma=3244548117
dF=2048, k=3, d=19110, d2=11, i0=3
Expected number of curves to find a factor of n digits:
35 40 45 50 55 60 65 70 75 80
4911 70940 1226976 2.5e+07 5.8e+08 1.6e+10 2.7e+13 4e+18 5.4e+23 Inf
Step 1 took 230ms
Using 10 small primes for NTT
Estimated memory usage: 4040K
Initializing tables of differences for F took 0ms
Computing roots of F took 9ms
Building F from its roots took 16ms
Computing 1/F took 9ms
Initializing table of differences for G took 0ms
Computing roots of G took 8ms
Building G from its roots took 16ms
Computing roots of G took 7ms
Building G from its roots took 16ms
Computing G * H took 6ms
Reducing G * H mod F took 5ms
Computing roots of G took 7ms
Building G from its roots took 17ms
Computing G * H took 5ms
Reducing G * H mod F took 5ms
Computing polyeval(F,G) took 34ms
Computing product of all F(g_i) took 0ms
Step 2 took 164ms
Expected time to find a factor of n digits:
35 40 45 50 55 60 65 70 75 80
32.25m 7.76h 5.60d 114.21d 7.27y 196.42y 337811y 5e+10y 7e+15y Inf

Expected curves: 4911, Expected time: 32.25m

```


INTERFACE TO GAP

Sage provides an interface to the GAP system. This system provides extensive group theory, combinatorics, etc.

The GAP interface will only work if GAP is installed on your computer; this should be the case, since GAP is included with Sage. The interface offers three pieces of functionality:

1. `gap_console()` - A function that dumps you into an interactive command-line GAP session.
2. `gap(expr)` - Evaluation of arbitrary GAP expressions, with the result returned as a string.
3. `gap.new(expr)` - Creation of a Sage object that wraps a GAP object. This provides a Pythonic interface to GAP. For example, if `f=gap.new(10)`, then `f.Factors()` returns the prime factorization of 10 computed using GAP.

9.1 First Examples

We factor an integer using GAP:

```
sage: n = gap(20062006); n
20062006
sage: n.parent()
Gap
sage: fac = n.Factors(); fac
[ 2, 17, 59, 73, 137 ]
sage: fac.parent()
Gap
sage: fac[1]
2
```

9.2 GAP and Singular

This example illustrates conversion between Singular and GAP via Sage as an intermediate step. First we create and factor a Singular polynomial.

```
sage: singular(389)
389
sage: R1 = singular.ring(0, '(x,y)', 'dp')
sage: f = singular('9*x^16-18*x^13*y^2-9*x^12*y^3+9*x^10*y^4-18*x^11*y^2+36*x^8*y^4+18*x^
↪ 7*y^5-18*x^5*y^6+9*x^6*y^4-18*x^3*y^6-9*x^2*y^7+9*y^8')
sage: F = f.factorize()
sage: print(F)
```

(continues on next page)

OUTPUT: element of F

EXAMPLES:

```

sage: x = gap('Z(13)')
sage: F = GF(13, 'a')
sage: F(x)
2
sage: F(gap('0*Z(13)'))
0
sage: F = GF(13^2, 'a')
sage: x = gap('Z(13)')
sage: F(x)
2
sage: x = gap('Z(13^2)^3')
sage: F(x)
12*a + 11
sage: F.multiplicative_generator()^3
12*a + 11

```

AUTHOR:

- David Joyner and William Stein

`sage.interfaces.gap.intmod_gap_to_sage(x)`

INPUT:

- x – Gap integer mod ring element

EXAMPLES:

```

sage: a = gap(Mod(3, 18)); a
ZmodnZObj( 3, 18 )
sage: b = sage.interfaces.gap.intmod_gap_to_sage(a); b
3
sage: b.parent()
Ring of integers modulo 18

sage: a = gap(Mod(3, 17)); a
Z(17)
sage: b = sage.interfaces.gap.intmod_gap_to_sage(a); b
3
sage: b.parent()
Finite Field of size 17

sage: a = gap(Mod(0, 17)); a
0*Z(17)
sage: b = sage.interfaces.gap.intmod_gap_to_sage(a); b
0
sage: b.parent()
Finite Field of size 17

sage: a = gap(Mod(3, 65537)); a
ZmodpZObj( 3, 65537 )
sage: b = sage.interfaces.gap.intmod_gap_to_sage(a); b
3

```

(continues on next page)

(continued from previous page)

```
sage: b.parent()  
Ring of integers modulo 65537
```

`sage.interfaces.gap.is_GapElement(x)`
Returns True if x is a GapElement.

EXAMPLES:

```
sage: from sage.interfaces.gap import is_GapElement  
sage: is_GapElement(gap(2))  
True  
sage: is_GapElement(2)  
False
```

`sage.interfaces.gap.reduce_load_GAP()`
Returns the GAP interface object defined in `sage.interfaces.gap`.

EXAMPLES:

```
sage: from sage.interfaces.gap import reduce_load_GAP  
sage: reduce_load_GAP()  
Gap
```


(continued from previous page)

```

##          Version 3          #
###         Release 4.4        #
## #       18 Apr 97          #
## #
## # Alice Niemeyer, Werner Nickel, Martin Schoenert
## # Johannes Meier, Alex Wegner,   Thomas Bischops
## # Frank Celler, Juergen Mnich,   Udo Polis
### ## Thomas Breuer, Goetz Pfeiffer, Hans U. Besche
##### Volkmar Felsch, Heiko Theissen, Alexander Hulpke
                          Ansgar Kaup, Akos Seress, Erzsebet Horvath
                          Bettina Eick
                          For help enter: ?<return>
gap>
```

`sage.interfaces.gap3.gap3_version()`

Return the version of GAP3 that you have in your PATH on your computer.

EXAMPLES:

```

sage: gap3_version() # random, optional -_
↪gap3
'lib: v3r4p4 1997/04/18, src: v3r4p0 1994/07/10, sys: usg gcc ansi'
```

INTERFACE TO GROEBNER FAN

AUTHOR:

- Anders Nedergaard Jensen: Write gfan C++ program, which implements algorithms many of which were invented by Jensen, Komei Fukuda, and Rekha Thomas.
- William Stein (2006-03-18): wrote gfan interface (first version)
- Marshall Hampton (2008-03-17): modified to use gfan-0.3, subprocess instead of os.popen2

TODO – much functionality of gfan-0.3 is still not exposed:

```
* at most 52 variables:
    - use gfan_substitute to make easier (?)
    MH: I think this is now irrelevant since gfan can accept the original ring variables

* --symmetry is really useful
    - permutations are 0-based *not* cycle notation; a <---> 0
    output is broken up much more nicely.

* -- can work in Z/pZ for p <= 32749

* -- can compute individual GB's for lex and revlex (via buchberger)
```

```
class sage.interfaces.gfan.Gfan
```

```
    Bases: object
```

```
    Interface to Anders Jensen's Groebner Fan program.
```


(continued from previous page)

```
True
sage: is_GpElement(2)
False
```

`sage.interfaces.gp.reduce_load_GP()`

Returns the GP interface object defined in `sage.interfaces.gp`.

EXAMPLES:

```
sage: from sage.interfaces.gp import reduce_load_GP
sage: reduce_load_GP()
PARI/GP interpreter
```


INTERFACE FOR EXTRACTING DATA AND GENERATING IMAGES FROM JMOL READABLE FILES.

JmolData is a no GUI version of Jmol useful for extracting data from files Jmol reads and for generating image files.

AUTHORS:

- Jonathan Gutow (2012-06-14): complete doctest coverage
- Jonathan Gutow (2012-03-21): initial version

```
class sage.interfaces.jmoldata.JmolData
    Bases: sage.structure.sage_object.SageObject
```

Todo: Create an animated image file (GIF) if spin is on and put data extracted from a file into a variable/string/structure to return

```
export_image(targetfile, datafile, datafile_cmd='script', image_type='PNG', figsize=5, **kwds)
    This executes JmolData.jar to make an image file.
```

INPUT:

- targetfile – the full path to the file where the image should be written.
- datafile – full path to the data file Jmol can read or text of a script telling Jmol what to read or load. If it is a script and the platform is cygwin, the filenames in the script should be in native windows format.
- datafile_cmd – (default 'script') 'load' or 'script' should be "load" for a data file.
- image_type – (default "PNG") 'PNG' 'JPG' or 'GIF'
- figsize – number (default 5) equal to (pixels/side)/100

OUTPUT:

Image file, .png, .gif or .jpg (default .png)

Note: Examples will generate an error message if a functional Java Virtual Machine (JVM) is not installed on the machine the Sage instance is running on.

Warning: Programmers using this module should check that the JVM is available before making calls to avoid the user getting error messages. Check for the JVM using the function `is_jvm_available()`, which returns True if a JVM is available.

EXAMPLES:

Use Jmol to load a pdb file containing some DNA from a web data base and make an image of the DNA. If you execute this in the notebook, the image will appear in the output cell:

```
sage: from sage.interfaces.jmoldata import JmolData
sage: JData = JmolData()
sage: script = "load =1lcd;display DNA;moveto 0.0 { -473 -713 -518 59.94} 100.0_
↳0.0 0.0 {21.17 26.72 27.295} 27.544636 {0.0 0.0 0.0} -25.287832 64.8414 0.0;"
sage: testfile = tmp_filename(ext="DNA.png")
sage: JData.export_image(targetfile=testfile,datafile=script,image_type="PNG")
↳# optional -- java internet
sage: print(os.path.exists(testfile)) # optional -- java internet
True
```

Use Jmol to save an image of a 3-D object created in Sage. This method is used internally by plot3d to generate static images. This example doesn't have correct scaling:

```
sage: from sage.interfaces.jmoldata import JmolData
sage: JData = JmolData()
sage: D = dodecahedron()
sage: from tempfile import NamedTemporaryFile
sage: archive = NamedTemporaryFile(suffix=".zip")
sage: D.export_jmol(archive.name)
sage: archive_native = archive.name
sage: import sys
sage: if sys.platform == 'cygwin':
.....:     import cygwin
.....:     archive_native = cygwin.cygpath(archive_native, 'w')
sage: script = f'set defaultdirectory "{archive_native}"\n'
sage: script += 'script SCRIPT\n'
sage: with NamedTemporaryFile(suffix=".png") as testfile: # optional -- java
.....:     JData.export_image(targetfile=testfile.name,
.....:                       datafile=script,
.....:                       image_type="PNG")
.....:     os.path.exists(testfile.name)
True
sage: archive.close()
```

is_jvm_available()

Returns True if the Java Virtual Machine is available and False if not.

EXAMPLES:

Check that it returns a boolean:

```
sage: from sage.interfaces.jmoldata import JmolData
sage: JData = JmolData()
sage: type(JData.is_jvm_available())
<... 'bool'>
```

INTERFACE TO KASH

Sage provides an interface to the KASH computer algebra system, which is a *free* (as in beer!) but *closed source* program for algebraic number theory that shares much common code with Magma. To use KASH, you must first install it. Visit its web page: <http://page.math.tu-berlin.de/~kant/kash.html>

Todo: Update the following sentence.

It is not enough to just have KASH installed on your computer.

The KASH interface offers three pieces of functionality:

1. `kash_console()` - A function that dumps you into an interactive command-line KASH session. Alternatively, type `!kash` from the Sage prompt.
2. `kash(expr)` - Creation of a Sage object that wraps a KASH object. This provides a Pythonic interface to KASH. For example, if `f=kash.new(10)`, then `f.Factors()` returns the prime factorization of 10 computed using KASH.
3. `kash.function_name(args ...)` - Call the indicated KASH function with the given arguments and return the result as a KASH object.
4. `kash.eval(expr)` - Evaluation of arbitrary KASH expressions, with the result returned as a string.

16.1 Issues

For some reason hitting Control-C to interrupt a calculation doesn't work correctly. (TODO)

16.2 Tutorial

The examples in this tutorial require that kash be installed.

16.2.1 Basics

Basic arithmetic is straightforward. First, we obtain the result as a string.

```
sage: kash.eval('(9 - 7) * (5 + 6)')           # optional -- kash
'22'
```

Next we obtain the result as a new KASH object.

```
sage: a = kash('(9 - 7) * (5 + 6)'); a       # optional -- kash
22
sage: a.parent()                             # optional -- kash
Kash
```

We can do arithmetic and call functions on KASH objects:

```
sage: a*a                                     # optional -- kash
484
sage: a.Factorial()                           # optional -- kash
11240000727777607680000
```

16.2.2 Integrated Help

Use the `kash.help(name)` command to get help about a given command. This returns a list of help for each of the definitions of `name`. Use `print kash.help(name)` to nicely print out all signatures.

16.2.3 Arithmetic

Using the `kash.new` command we create Kash objects on which one can do arithmetic.

```
sage: a = kash(12345)                         # optional -- kash
sage: b = kash(25)                            # optional -- kash
sage: a/b                                     # optional -- kash
2469/5
sage: a**b                                    # optional -- kash
1937659030411463935651167391656422626577614411586152317674869233464019922771432158872187137603759765625
```

16.2.4 Variable assignment

Variable assignment using `kash` is takes place in Sage.

```
sage: a = kash('32233')                       # optional -- kash
sage: a                                       # optional -- kash
32233
```

In particular, `a` is not defined as part of the KASH session itself.

```
sage: kash.eval('a')                          # optional -- kash
"Error, the variable 'a' must have a value"
```

Use `a.name()` to get the name of the KASH variable:

(continued from previous page)

```

sage: m5 = matrix(ZZ, 2, 3, [2, 2, 2, -1, -1, -1])
sage: sage_chcm = ChainComplex({1: m1, 4: m4, 5: m5}, degree = -1) # optional_
↪- kenzo
sage: kenzo_chcm = KChainComplex(sage_chcm) # optional_
↪- kenzo
sage: kenzo_chcm # optional_
↪- kenzo
[K... Chain-Complex]
sage: differential_morphism = kenzo_chcm.differential() # optional_
↪- kenzo
sage: differential_morphism # optional_
↪- kenzo
[K... Morphism (degree -1): K... -> K...]
sage: differential_morphism.source_complex() # optional_
↪- kenzo
[K... Chain-Complex]
    
```

abstract (*object=None*)

Return a morphism, difference of the morphism `self` and the morphism(s) given by the parameter `object`.

INPUT:

- `object` – A `KenzoChainComplexMorphism` instance, a tuple of `KenzoChainComplexMorphism` instances or `None` (default).

OUTPUT:

- A *`KenzoChainComplexMorphism`*, difference of the morphism `self` and the morphism(s) given by `object` (if `object` is `None`, `self` morphism is returned). For example, if `object = (mrph1, mrph2, mrph3)` the result is `self - mrph1 - mrph2 - mrph3`.

EXAMPLES:

```

sage: from sage.interfaces.kenzo import KChainComplex # optional_
↪- kenzo
sage: m1 = matrix(ZZ, 3, 2, [-1, 1, 3, -4, 5, 6])
sage: m4 = matrix(ZZ, 2, 2, [1, 2, 3, 6])
sage: m5 = matrix(ZZ, 2, 3, [2, 2, 2, -1, -1, -1])
sage: sage_chcm = ChainComplex({1: m1, 4: m4, 5: m5}, degree = -1) # optional_
↪- kenzo
sage: kenzo_chcm = KChainComplex(sage_chcm) # optional_
↪- kenzo
sage: kenzo_chcm # optional_
↪- kenzo
[K... Chain-Complex]
sage: idnt = kenzo_chcm.identity_morphism() # optional_
↪- kenzo
sage: idnt # optional_
↪- kenzo
[K... Morphism (degree 0): K... -> K...]
sage: opps_id = idnt.opposite() # optional_
↪- kenzo
sage: opps_id # optional_
↪- kenzo
[K... Morphism (degree 0): K... -> K...]
    
```

(continues on next page)

(continued from previous page)

```

sage: null = kenzo_chcm.null_morphism() # optional_
↪- kenzo
sage: null # optional_
↪- kenzo
[K... Morphism (degree 0): K... -> K...]
sage: idx2 = idnt.substract(oppo_id) # optional_
↪- kenzo
sage: oppo_idx2 = idx2.substract\
.....: ((oppo_id, idnt, idnt, null, idx2.substract(oppo_id))) # optional_
↪- kenzo
sage: kenzo_chcm.basis(4) # optional_
↪- kenzo
['G4G0', 'G4G1']
sage: idx2.evaluation(4, [2, 'G4G0', -5, 'G4G1']) # optional_
↪- kenzo

-----{CMBN 4}
<4 * G4G0>
<-10 * G4G1>
-----

sage: oppo_idx2.evaluation(4, [2, 'G4G0', -5, 'G4G1']) # optional_
↪- kenzo

-----{CMBN 4}
<-4 * G4G0>
<10 * G4G1>
-----

```

sum(*object=None*)

Return a morphism, sum of the morphism `self` and the morphism(s) given by the parameter `object`.

INPUT:

- `object` – A *KenzoChainComplexMorphism* instance, a tuple of *KenzoChainComplexMorphism* instances or `None` (default).

OUTPUT:

- A *KenzoChainComplexMorphism*, sum of the morphism `self` and the morphism(s) given by `object` (if `object` is `None`, `self` morphism is returned).

EXAMPLES:

```

sage: from sage.interfaces.kenzo import KChainComplex # optional_
↪- kenzo
sage: m1 = matrix(ZZ, 3, 2, [-1, 1, 3, -4, 5, 6])
sage: m4 = matrix(ZZ, 2, 2, [1, 2, 3, 6])
sage: m5 = matrix(ZZ, 2, 3, [2, 2, 2, -1, -1])
sage: sage_chcm = ChainComplex({1: m1, 4: m4, 5: m5}, degree = -1) # optional_
↪- kenzo
sage: kenzo_chcm = KChainComplex(sage_chcm) # optional_
↪- kenzo
sage: kenzo_chcm # optional_
↪- kenzo

```

(continues on next page)

(continued from previous page)

```

[K... Chain-Complex]
sage: idnt = kenzo_chcm.identity_morphism() # optional_
↪- kenzo
sage: idnt # optional_
↪- kenzo
[K... Morphism (degree 0): K... -> K...]
sage: opps_id = idnt.opposite() # optional_
↪- kenzo
sage: opps_id # optional_
↪- kenzo
[K... Morphism (degree 0): K... -> K...]
sage: null = kenzo_chcm.null_morphism() # optional_
↪- kenzo
sage: null # optional_
↪- kenzo
[K... Morphism (degree 0): K... -> K...]
sage: idx2 = idnt.sum(idnt) # optional_
↪- kenzo
sage: idx5 = idx2.sum(\
....: (opps_id, idnt, idnt, null, idx2.sum(idnt), opps_id)) # optional_
↪- kenzo
sage: kenzo_chcm.basis(4) # optional_
↪- kenzo
['G4G0', 'G4G1']
sage: idx2.evaluation(4, [2, 'G4G0', -5, 'G4G1']) # optional_
↪- kenzo

-----{CMBN 4}
<4 * G4G0>
<-10 * G4G1>
-----

sage: idx5.evaluation(4, [2, 'G4G0', -5, 'G4G1']) # optional_
↪- kenzo

-----{CMBN 4}
<10 * G4G0>
<-25 * G4G1>
-----

```

target_complex()

Return the target chain complex of the morphism.

OUTPUT:

- A *KenzoChainComplex*

EXAMPLES:

```

sage: from sage.interfaces.kenzo import KChainComplex # optional_
↪- kenzo
sage: m1 = matrix(ZZ, 3, 2, [-1, 1, 3, -4, 5, 6])
sage: m4 = matrix(ZZ, 2, 2, [1, 2, 3, 6])
sage: m5 = matrix(ZZ, 2, 3, [2, 2, 2, -1, -1, -1])

```

(continues on next page)

(continued from previous page)

```
Chain complex with at most 3 nonzero terms over Integer Ring
sage: C._ascii_art_() # optional - kenzo
0 <-- C_4 <-- 0 ... 0 <-- C_0 <-- 0
sage: [C.homology(i) for i in range(6)] # optional - kenzo
[Z, 0, 0, 0, Z, 0]
```

`sage.interfaces.kenzo.SFiniteSimplicialSet(ksimpset, limit)`

Convert the limit-skeleton of a finite simplicial set in Kenzo to a finite `SimplicialSet` in Sage.

INPUT:

- `ksimpset` – A finite simplicial set in Kenzo.
- `limit` – A natural number.

OUTPUT:

- A finite `SimplicialSet`.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import SimplicialSet
sage: from sage.interfaces.kenzo import AbstractSimplex, \
.....: KFiniteSimplicialSet, SFiniteSimplicialSet, Sphere # optional - kenzo
sage: s0 = AbstractSimplex(0, name='s0') # optional - kenzo
sage: s1 = AbstractSimplex(0, name='s1') # optional - kenzo
sage: s2 = AbstractSimplex(0, name='s2') # optional - kenzo
sage: s01 = AbstractSimplex(1, name='s01') # optional - kenzo
sage: s02 = AbstractSimplex(1, name='s02') # optional - kenzo
sage: s12 = AbstractSimplex(1, name='s12') # optional - kenzo
sage: s012 = AbstractSimplex(2, name='s012') # optional - kenzo
sage: Triangle = SimplicialSet({s01: (s1, s0), \
.....: s02: (s2, s0), s12: (s2, s1)}, base_point = s0) # optional - kenzo
sage: KTriangle = KFiniteSimplicialSet(Triangle) # optional - kenzo
sage: STriangle = SFiniteSimplicialSet(KTriangle, 1) # optional - kenzo
sage: STriangle.homology() # optional - kenzo
{0: 0, 1: Z}
sage: S1 = simplicial_sets.Sphere(1) # optional - kenzo
sage: S3 = simplicial_sets.Sphere(3) # optional - kenzo
sage: KS1vS3 = KFiniteSimplicialSet(S1.wedge(S3)) # optional - kenzo
sage: SS1vS3 = SFiniteSimplicialSet(KS1vS3, 3) # optional - kenzo
sage: SS1vS3.homology() # optional - kenzo
{0: 0, 1: Z, 2: 0, 3: Z}
```

`sage.interfaces.kenzo.Sphere(n)`

Return the n dimensional sphere as a Kenzo simplicial set.

INPUT:

- `n` – the dimension of the sphere

OUTPUT:

- A *KenzoSimplicialSet*

EXAMPLES:

```
sage: from sage.interfaces.kenzo import Sphere # optional - kenzo
sage: s2 = Sphere(2) # optional - kenzo
sage: s2 # optional - kenzo
[K1 Simplicial-Set]
sage: [s2.homology(i) for i in range(8)] # optional - kenzo
[Z, 0, Z, 0, 0, 0, 0, 0]
```

`sage.interfaces.kenzo.build_morphism`(*source_complex*, *target_complex*, *degree*, *algorithm*, *strategy*, *orgn*)
Build a morphism of chain complexes by means of the corresponding build-mrph Kenzo function.

INPUT:

- *source_complex* – The source object as a KenzoChainComplex instance
- *target_complex* – The target object as a KenzoChainComplex instance
- *degree* – An integer number representing the degree of the morphism
- *algorithm* – A Lisp function defining the mapping (:intr slot in Kenzo)
- *strategy* – The strategy (:strt slot in Kenzo), which must be one of the two strings `gnrt` or `cmbn`, depending if the *algorithm* (a Lisp function) uses as arguments a degree and a generator or a combination, respectively.
- *orgn* – A list containing a description about the origin of the morphism

OUTPUT:

- A *KenzoChainComplexMorphism*

EXAMPLES:

```
sage: from sage.interfaces.kenzo import KenzoChainComplex,\
.....: build_morphism # optional - kenzo
sage: from sage.libs.ecl import ecl_eval
sage: ZCC = KenzoChainComplex(ecl_eval("(z-chcm"))) # optional - kenzo
sage: A = build_morphism(ZCC, ZCC, -1,\
.....: ecl_eval("#'(lambda (comb) (cmbn (1- (degr comb))))"),\
.....: "cmbn", ["zero morphism on ZCC"]) # optional - kenzo
sage: A.target_complex() # optional - kenzo
[K... Chain-Complex]
sage: A.degree() # optional - kenzo
-1
sage: type(A) # optional - kenzo
<class 'sage.interfaces.kenzo.KenzoChainComplexMorphism'>
```

`sage.interfaces.kenzo.k2s_matrix`(*kmatrix*)
Convert an array of ECL to a matrix of Sage.

INPUT:

- *kmatrix* – An array in ECL

EXAMPLES:

```
sage: from sage.interfaces.kenzo import k2s_matrix # optional - kenzo
sage: from sage.libs.ecl import EclObject
sage: M = EclObject("#2A((1 2 3) (3 2 1) (1 1 1))")
sage: k2s_matrix(M) # optional - kenzo
```

(continues on next page)

(continued from previous page)

```
[1 2 3]
[3 2 1]
[1 1 1]
```

`sage.interfaces.kenzo.morphism_dictmat`(*morphism*)
 Computes a list of matrices in ECL associated to a morphism in Sage.

INPUT:

- *morphism* – A morphism of chain complexes

OUTPUT:

- A `EclObject`

EXAMPLES:

```
sage: from sage.interfaces.kenzo import morphism_dictmat # optional - kenzo
sage: X = simplicial_complexes.Simplex(1)
sage: Y = simplicial_complexes.Simplex(0)
sage: g = Hom(X,Y)({0:0, 1:0})
sage: f = g.associated_chain_complex_morphism()
sage: morphism_dictmat(f) # optional - kenzo
<ECL: ((2 . #2A()) (1 . #2A()) (0 . #2A((1 1))))>
```

`sage.interfaces.kenzo.pairing`(*slist*)
 Convert a list of Sage (which has an even length) to an assoc list in ECL.

INPUT:

- *slist* – A list in Sage

OUTPUT:

- A `EclObject`

EXAMPLES:

```
sage: from sage.interfaces.kenzo import pairing # optional - kenzo
sage: l = [1,2,3]
sage: pairing(l) # optional - kenzo
<ECL: ((2 . 3))>
```

`sage.interfaces.kenzo.s2k_dictmat`(*sdictmat*)
 Convert a dictionary in Sage, whose values are matrices, to an assoc list in ECL.

INPUT:

- *sdictmat* – A dictionary in Sage

OUTPUT:

- A `EclObject`

EXAMPLES:

```
sage: from sage.interfaces.kenzo import s2k_dictmat # optional - kenzo
sage: A = Matrix([[1,2,3],[3,2,1],[1,1,1]])
sage: B = Matrix([[1,2],[2,1],[1,1]])
sage: d = {1 : A, 2 : B}
```

(continues on next page)

(continued from previous page)

```
sage: s2k_dictmat(d) # optional - kenzo
<ECL: ((2 . #2A((1 2) (2 1) (1 1))) (1 . #2A((1 2 3) (3 2 1) (1 1 1))))>
```

`sage.interfaces.kenzo.s2k_listofmorphisms()`

Computes a list of morphisms of chain complexes in Kenzo from a list of morphisms in Sage.

INPUT:

- `l` – A list of morphisms of chain complexes

OUTPUT:

- A `EclObject`

EXAMPLES:

```
sage: from sage.interfaces.kenzo import s2k_listofmorphisms # optional - kenzo
sage: C1 = ChainComplex({1: matrix(ZZ, 0, 2, [])}, degree_of_differential=-1)
sage: C2 = ChainComplex({1: matrix(ZZ, 1, 2, [1, 0])}, degree_of_differential=-1)
sage: C3 = ChainComplex({0: matrix(ZZ, 0, 2, []), degree_of_differential=-1)
sage: M1 = Hom(C2, C1)({1: matrix(ZZ, 2, 2, [2, 0, 0, 2])})
sage: M2 = Hom(C3, C2)({0: matrix(ZZ, 1, 2, [2, 0])})
sage: l = [M1, M2]
sage: s2k_listofmorphisms(l) # optional - kenzo
<ECL: ([K... Morphism (degree 0): K... -> K... [K... Morphism (degree 0): K... ->
->K...])>
```

`sage.interfaces.kenzo.s2k_matrix(smatrix)`

Convert a matrix of Sage to an array of ECL.

INPUT:

- `smatrix` – A matrix in Sage

OUTPUT:

- A `EclObject`

EXAMPLES:

```
sage: from sage.interfaces.kenzo import s2k_matrix # optional - kenzo
sage: A = Matrix([[1,2,3],[3,2,1],[1,1,1]])
sage: s2k_matrix(A) # optional - kenzo
<ECL: #2A((1 2 3) (3 2 1) (1 1 1))>
```

INTERFACE TO LATTE INTEGRALE PROGRAMS

```
sage.interfaces.latte.count(arg, ehrhart_polynomial=False, multivariate_generating_function=False,
    raw_output=False, verbose=False, **kwds)
```

Call to the program count from LattE integrale

INPUT:

- *arg* – a cdd or LattE description string
- *ehrhart_polynomial*, *multivariate_generating_function* – to compute Ehrhart polynomial or multivariate generating function instead of just counting points
- *raw_output* – if True then return directly the output string from LattE
- For all other options of the count program, consult the LattE manual

OUTPUT:

Either a string (if *raw_output* if set to True) or an integer (when counting points), or a polynomial (if *ehrhart_polynomial* is set to True) or a multivariate THING (if *multivariate_generating_function* is set to True)

EXAMPLES:

```
sage: from sage.interfaces.latte import count # optional - latte_int
sage: P = 2 * polytopes.cube()
```

Counting integer points from either the H or V representation:

```
sage: count(P.cdd_Hrepresentation(), cdd=True) # optional - latte_int
125
sage: count(P.cdd_Vrepresentation(), cdd=True) # optional - latte_int
125
```

Ehrhart polynomial:

```
sage: count(P.cdd_Hrepresentation(), cdd=True, ehrhart_polynomial=True) # optional -
↳- latte_int
64*t^3 + 48*t^2 + 12*t + 1
```

Multivariate generating function currently only work with *raw_output*=True:

```
sage: opts = {'cdd': True,
.....:       'multivariate_generating_function': True,
.....:       'raw_output': True}
sage: cddin = P.cdd_Hrepresentation()
```

(continues on next page)

(continued from previous page)

```
sage: print(count(cddin, **opts)) # optional - latte_int
x[0]^2*x[1]^(-2)*x[2]^(-2)/((1-x[1])*(1-x[2])*(1-x[0]^(-1)))
+ x[0]^(-2)*x[1]^(-2)*x[2]^(-2)/((1-x[1])*(1-x[2])*(1-x[0]))
+ x[0]^2*x[1]^(-2)*x[2]^2/((1-x[1])*(1-x[2]^(-1))*(1-x[0]^(-1)))
+ x[0]^(-2)*x[1]^(-2)*x[2]^2/((1-x[1])*(1-x[0])*(1-x[2]^(-1)))
+ x[0]^2*x[1]^2*x[2]^(-2)/((1-x[2])*(1-x[1]^(-1))*(1-x[0]^(-1)))
+ x[0]^(-2)*x[1]^2*x[2]^(-2)/((1-x[2])*(1-x[0])*(1-x[1]^(-1)))
+ x[0]^2*x[1]^2*x[2]^2/((1-x[2]^(-1))*(1-x[1]^(-1))*(1-x[0]^(-1)))
+ x[0]^(-2)*x[1]^2*x[2]^2/((1-x[0])*(1-x[2]^(-1))*(1-x[1]^(-1)))
```

```
sage.interfaces.latte.integrate(arg, polynomial=None, algorithm='triangulate', raw_output=False,
                               verbose=False, **kwds)
```

Call to the function `integrate` from `LattE integrale`.

INPUT:

- `arg` – a `cdd` or `LattE` description string.
- `polynomial` – multivariate polynomial or valid `LattE` polynomial description string. If given, the valuation parameter of `LattE` is set to `integrate`, and is set to `volume` otherwise.
- `algorithm` – (default: `'triangulate'`) the integration method. Use `'triangulate'` for polytope triangulation or `'cone-decompose'` for tangent cone decomposition method.
- `raw_output` – if `True` then return directly the output string from `LattE`.
- `verbose` – if `True` then return directly verbose output from `LattE`.
- For all other options of the `integrate` program, consult the `LattE manual`.

OUTPUT:

Either a string (if `raw_output` if set to `True`) or a rational.

EXAMPLES:

```
sage: from sage.interfaces.latte import integrate # optional - latte_int
sage: P = 2 * polytopes.cube()
sage: x, y, z = polygen(QQ, 'x, y, z')
```

Integrating over a polynomial over a polytope in either the H or V representation:

```
sage: integrate(P.cdd_Hrepresentation(), x^2*y^2*z^2, cdd=True) # optional -
↳ latte_int
4096/27
sage: integrate(P.cdd_Vrepresentation(), x^2*y^2*z^2, cdd=True) # optional -
↳ latte_int
4096/27
```

Computing the volume of a polytope in either the H or V representation:

```
sage: integrate(P.cdd_Hrepresentation(), cdd=True) # optional - latte_int
64
sage: integrate(P.cdd_Vrepresentation(), cdd=True) # optional - latte_int
64
```

Polynomials given as a string in `LattE` description are also accepted:

```
sage: integrate(P.cdd_Hrepresentation(), '[[1,[2,2,2]]]', cdd=True) # optional -  
↔ latte_int  
4096/27
```

`sage.interfaces.latte.to_latte_polynomial(polynomial)`
Helper function to transform a polynomial to its LattE description.

INPUT:

- `polynomial` – a multivariate polynomial.

OUTPUT:

A string that describes the monomials list and exponent vectors.

INTERFACE TO LIE

LiE is a software package under development at CWI since January 1988. Its purpose is to enable mathematicians and physicists to obtain on-line information as well as to interactively perform computations of a Lie group theoretic nature. It focuses on the representation theory of complex semisimple (reductive) Lie groups and algebras, and on the structure of their Weyl groups and root systems.

Type `lie. [tab]` for a list of all the functions available from your LiE install. Type `lie. [tab]?` for LiE's help about a given function. Type `lie(...)` to create a new LiE object, and `lie.eval(...)` to run a string using LiE (and get the result back as a string).

To access the LiE interpreter directly, run `lie_console()`.

EXAMPLES:

```
sage: a4 = lie('A4')           # optional - lie
sage: lie.diagram('A4')      # optional - lie
0---0---0---0
1  2  3  4
A4

sage: lie.diagram(a4)       # optional - lie
0---0---0---0
1  2  3  4
A4

sage: a4.diagram()          # optional - lie
0---0---0---0
1  2  3  4
A4

sage: a4.Cartan()           # optional - lie
[[ 2,-1, 0, 0]
,[-1, 2,-1, 0]
,[ 0,-1, 2,-1]
,[ 0, 0,-1, 2]
]
sage: lie.LR_tensor([3,1],[2,2]) # optional - lie
1X[5,3]
```

19.1 Tutorial

The following examples are taken from Section 2.1 of the LiE manual.

You can perform basic arithmetic operations in LiE.

```
sage: lie.eval('19+68') # optional - lie
'87'
sage: a = lie('1111111111*1111111111') # optional - lie
sage: a # optional - lie
1234567900987654321
sage: a/1111111111 # optional - lie
1111111111
sage: a = lie('345') # optional - lie
sage: a^2+3*a-5 # optional - lie
120055
sage: _ / 7*a # optional - lie
5916750
```

Vectors in LiE are created using square brackets. Notice that the indexing in LiE is 1-based, unlike Python/Sage which is 0-based.

```
sage: v = lie('[3,2,6873,-38]') # optional - lie
sage: v # optional - lie
[3,2,6873,-38]
sage: v[3] # optional - lie
6873
sage: v+v # optional - lie
[6,4,13746,-76]
sage: v*v # optional - lie
47239586
sage: v+234786 # optional - lie
[3,2,6873,-38,234786]
sage: v-3 # optional - lie
[3,2,-38]
sage: v^v # optional - lie
[3,2,6873,-38,3,2,6873,-38]
```

You can also work with matrices in LiE.

```
sage: m = lie('[[1,0,3,3],[12,4,-4,7],[-1,9,8,0],[3,-5,-2,9]]') # optional - lie
sage: m # optional - lie
[[ 1,  0,  3,  3]
 , [12,  4, -4,  7]
 , [-1,  9,  8,  0]
 , [ 3, -5, -2,  9]
 ]
sage: print(lie.eval('*'+m._name)) # optional - lie
[[1,12,-1, 3]
 , [0, 4, 9,-5]
 , [3,-4, 8,-2]
 , [3, 7, 0, 9]
 ]
```

(continues on next page)

(continued from previous page)

```
sage: m^3 # optional - lie
[[ 220, 87, 81, 375]
, [-168, -1089, 13, 1013]
, [1550, 357, -55, 1593]
, [-854, -652, 98, -170]
]
sage: v*m # optional - lie
[-6960, 62055, 55061, -319]
sage: m*v # optional - lie
[20508, -27714, 54999, -14089]
sage: v*m*v # optional - lie
378549605
sage: m+v # optional - lie
[[ 1, 0, 3, 3]
, [12, 4, -4, 7]
, [-1, 9, 8, 0]
, [ 3, -5, -2, 9]
, [ 3, 2, 6873, -38]
]
sage: m-2 # optional - lie
[[ 1, 0, 3, 3]
, [-1, 9, 8, 0]
, [ 3, -5, -2, 9]
]
```

LiE handles multivariate (Laurent) polynomials.

```
sage: lie('X[1,2]') # optional - lie
1X[1,2]
sage: -3*_ # optional - lie
-3X[1,2]
sage: _ + lie('4X[-1,4]') # optional - lie
4X[-1,4] - 3X[ 1,2]
sage: _^2 # optional - lie
16X[-2,8] - 24X[ 0,6] + 9X[ 2,4]
sage: lie('(4X[-1,4]-3X[1,2])*(X[2,0]-X[0,-4])') # optional - lie
-4X[-1, 0] + 3X[ 1,-2] + 4X[ 1, 4] - 3X[ 3, 2]
sage: _ - _ # optional - lie
0X[0,0]
```

You can call LiE's built-in functions using `lie.functionname`.

```
sage: lie.partitions(6) # optional - lie
[[6,0,0,0,0,0]
, [5,1,0,0,0,0]
, [4,2,0,0,0,0]
, [4,1,1,0,0,0]
, [3,3,0,0,0,0]
, [3,2,1,0,0,0]
, [3,1,1,1,0,0]
, [2,2,2,0,0,0]
```

(continues on next page)

(continued from previous page)

```

    , [2, 2, 1, 1, 0, 0]
    , [2, 1, 1, 1, 1, 0]
    , [1, 1, 1, 1, 1, 1]
  ]
sage: lie.diagram('E8') # optional - lie
      0 2
      |
      |
0---0---0---0---0---0
1  3  4  5  6  7  8
E8

```

You can define your own functions in LiE using `lie.eval`. Once you've defined a function (say `f`), you can call it using `lie.f`; however, user-defined functions do not show up when using tab-completion.

```

sage: lie.eval('f(int x) = 2*x') # optional - lie
''
sage: lie.f(984) # optional - lie
1968
sage: lie.eval('f(int n) = a=3*n-7; if a < 0 then a = -a fi; 7^a+a^3-4*a-57') # optional
↔- lie
''
sage: lie.f(2) # optional - lie
-53
sage: lie.f(5) # optional - lie
5765224

```

LiE's help can be accessed through `lie.help('functionname')` where `functionname` is the function you want to receive help for.

```

sage: print(lie.help('diagram')) # optional - lie
diagram(g). Prints the Dynkin diagram of g, also indicating
the type of each simple component printed, and labeling the nodes as
done by Bourbaki (for the second and further simple components the
labels are given an offset so as to make them disjoint from earlier
labels). The labeling of the vertices of the Dynkin diagram prescribes
the order of the coordinates of root- and weight vectors used in LiE.

```

This can also be accessed with `lie.functionname?`.

With the exception of groups, all LiE data types can be converted into native Sage data types by calling the `.sage()` method.

Integers:

```

sage: a = lie('1234') # optional - lie
sage: b = a.sage(); b # optional - lie
1234
sage: type(b) # optional - lie
<class 'sage.rings.integer.Integer'>

```

Vectors:

```

sage: a = lie('[1,2,3]') # optional - lie
sage: b = a.sage(); b # optional - lie
[1, 2, 3]
sage: type(b) # optional - lie
<... 'list'>

```

Matrices:

```

sage: a = lie('[[1,2],[3,4]]') # optional - lie
sage: b = a.sage(); b # optional - lie
[1 2]
[3 4]
sage: type(b) # optional - lie
<class 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>

```

Polynomials:

```

sage: a = lie('X[1,2] - 2*X[2,1]') # optional - lie
sage: b = a.sage(); b # optional - lie
-2*x0^2*x1 + x0*x1^2
sage: type(b) # optional - lie
<class 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>

```

Text:

```

sage: a = lie('"text"') # optional - lie
sage: b = a.sage(); b # optional - lie
'text'
sage: type(b) # optional - lie
<... 'str'>

```

LiE can be programmed using the Sage interface as well. Section 5.1.5 of the manual gives an example of a function written in LiE's language which evaluates a polynomial at a point. Below is a (roughly) direct translation of that program into Python / Sage.

```

sage: def eval_pol(p, pt): # optional - lie
.....:     s = 0
.....:     for i in range(1,p.length().sage()+1):
.....:         m = 1
.....:         for j in range(1,pt.size().sage()+1):
.....:             m *= pt[j]^p.expon(i)[j]
.....:         s += p.coef(i)*m
.....:     return s
sage: a = lie('X[1,2]') # optional - lie
sage: b1 = lie('[1,2]') # optional - lie
sage: b2 = lie('[2,3]') # optional - lie
sage: eval_pol(a, b1) # optional - lie
4
sage: eval_pol(a, b2) # optional - lie
18

```

AUTHORS:

- Mike Hansen 2007-08-27

- William Stein (template)

```
class sage.interfaces.lie.LiE(maxread=None, script_subdirectory=None, logfile=None, server=None)
  Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.expect.Expect
```

Interface to the LiE interpreter.

Type `lie. [tab]` for a list of all the functions available from your LiE install. Type `lie. [tab]?` for LiE's help about a given function. Type `lie(...)` to create a new LiE object, and `lie.eval(...)` to run a string using LiE (and get the result back as a string).

console()

Spawn a new LiE command-line session.

EXAMPLES:

```
sage: lie.console() # not tested
LiE version 2.2.2 created on Sep 26 2007 at 18:13:19
Authors: Arjeh M. Cohen, Marc van Leeuwen, Bert Lisser.
Free source code distribution
...
```

eval(code, strip=True, **kwds)

EXAMPLES:

```
sage: lie.eval('2+2') # optional - lie
'4'
```

function_call(function, args=None, kwds=None)

EXAMPLES:

```
sage: lie.function_call("diagram", args=['A4']) # optional - lie
0---0---0---0
1  2  3  4
A4
```

get(var)

Get the value of the variable `var`.

EXAMPLES:

```
sage: lie.set('x', '2') # optional - lie
sage: lie.get('x') # optional - lie
'2'
```

get_using_file(var)

EXAMPLES:

```
sage: lie.get_using_file('x')
Traceback (most recent call last):
...
NotImplementedError
```

help(command)

Return a string of the LiE help for `command`.

EXAMPLES:

```
sage: lie.help('diagram') # optional - lie
'diagram(g)...'
```

read(filename)

EXAMPLES:

```
sage: filename = tmp_filename()
sage: with open(filename, 'w') as f:
....:     _ = f.write('x = 2\n')
sage: lie.read(filename) # optional - lie
sage: lie.get('x')      # optional - lie
'2'
sage: import os
sage: os.unlink(filename)
```

set(var, value)

Set the variable var to the given value.

EXAMPLES:

```
sage: lie.set('x', '2') # optional - lie
sage: lie.get('x')     # optional - lie
'2'
```

version()

EXAMPLES:

```
sage: lie.version() # optional - lie
'2.2'
```

class sage.interfaces.lie.LiEElement(parent, value, is_name=False, name=None)

Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.expect.ExpectElement

type()

EXAMPLES:

```
sage: m = lie('[[[1,0,3,3],[12,4,-4,7],[-1,9,8,0],[3,-5,-2,9]]]') # optional - lie
sage: m.type() # optional - lie
'mat'
```

class sage.interfaces.lie.LiEFunction(parent, name)

Bases: sage.interfaces.expect.ExpectFunction

class sage.interfaces.lie.LiEFunctionElement(obj, name)

Bases: sage.interfaces.expect.FunctionElement

sage.interfaces.lie.is_LiEElement(x)

EXAMPLES:

```
sage: from sage.interfaces.lie import is_LiEElement
sage: l = lie(2) # optional - lie
sage: is_LiEElement(l) # optional - lie
True
sage: is_LiEElement(2)
False
```

`sage.interfaces.lie.lie_console()`
Spawn a new LiE command-line session.

EXAMPLES:

```
sage: from sage.interfaces.lie import lie_console
sage: lie_console() # not tested
LiE version 2.2.2 created on Sep 26 2007 at 18:13:19
Authors: Arjeh M. Cohen, Marc van Leeuwen, Bert Lisser.
Free source code distribution
...
```

`sage.interfaces.lie.lie_version()`

EXAMPLES:

```
sage: from sage.interfaces.lie import lie_version
sage: lie_version() # optional - lie
'2.2'
```

`sage.interfaces.lie.reduce_load_lie()`

EXAMPLES:

```
sage: from sage.interfaces.lie import reduce_load_lie
sage: reduce_load_lie()
LiE Interpreter
```

LISP INTERFACE

EXAMPLES:

```
sage: lisp.eval('(* 4 5)')
'20'
sage: a = lisp(3); b = lisp(5)
sage: a + b
8
sage: a * b
15
sage: a / b
3/5
sage: a - b
-2
sage: a.sin()
0.14112
sage: b.cos()
0.2836622
sage: a.exp()
20.085537
sage: lisp.eval('(+ %s %s)'%(a.name(), b.name()))
'8'
```

One can define functions and the interface supports object-oriented notation for calling them:

```
sage: lisp.eval('(defun factorial (n) (if (= n 1) 1 (* n (factorial (- n 1)))))')
'FACTORIAL'
sage: lisp('(factorial 10)')
3628800
sage: lisp(10).factorial()
3628800
sage: a = lisp(17)
sage: a.factorial()
355687428096000
```

AUTHORS: – William Stein (first version) – William Stein (2007-06-20): significant improvements.

class `sage.interfaces.lisp.Lisp`(*maxread=None, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None*)

Bases: `sage.interfaces.expect.Expect`

EXAMPLES:

```
sage: lisp == loads(dumps(lisp))
True
```

console()

Spawn a new Lisp command-line session.

EXAMPLES:

```
sage: lisp.console() #not tested
ECL (Embeddable Common-Lisp) ...
Copyright (C) 1984 Taiichi Yuasa and Masami Hagiya
Copyright (C) 1993 Giuseppe Attardi
Copyright (C) 2000 Juan J. Garcia-Ripoll
ECL is free software, and you are welcome to redistribute it
under certain conditions; see file 'Copyright' for details.
Type :h for Help. Top level.
...
```

eval(code, strip=True, **kwds)

EXAMPLES:

```
sage: lisp.eval('+ 2 2')
'4'
```

function_call(function, args=None, kwds=None)

Calls the Lisp function with given args and kwds. For Lisp functions, the kwds are ignored.

EXAMPLES:

```
sage: lisp.function_call('sin', ['2'])
0.9092974
sage: lisp.sin(2)
0.9092974
```

get(var)

EXAMPLES:

```
sage: lisp.set('x', '2')
sage: lisp.get('x')
'2'
```

help(command)

EXAMPLES:

```
sage: lisp.help('setq')
Traceback (most recent call last):
...
NotImplementedError
```

kill(var)

EXAMPLES:

```
sage: lisp.kill('x')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
NotImplementedError
```

set(var, value)

Set the variable var to the given value.

EXAMPLES:

```
sage: lisp.set('x', '2')
sage: lisp.get('x')
'2'
```

version()

Returns the version of Lisp being used.

EXAMPLES:

```
sage: lisp.version()
'Version information is given by lisp.console().'
```

class sage.interfaces.lisp.LispElement(*parent, value, is_name=False, name=None*)
 Bases: sage.structure.element.RingElement, sage.interfaces.expect.ExpectElement

class sage.interfaces.lisp.LispFunction(*parent, name*)
 Bases: sage.interfaces.expect.ExpectFunction

class sage.interfaces.lisp.LispFunctionElement(*obj, name*)
 Bases: sage.interfaces.expect.FunctionElement

sage.interfaces.lisp.is_LispElement(*x*)

EXAMPLES:

```
sage: from sage.interfaces.lisp import is_LispElement
sage: is_LispElement(lisp(2))
True
sage: is_LispElement(2)
False
```

sage.interfaces.lisp.lisp_console()

Spawn a new Lisp command-line session.

EXAMPLES:

```
sage: lisp.console() #not tested
ECL (Embeddable Common-Lisp) ...
Copyright (C) 1984 Taiichi Yuasa and Masami Hagiya
Copyright (C) 1993 Giuseppe Attardi
Copyright (C) 2000 Juan J. Garcia-Ripoll
ECL is free software, and you are welcome to redistribute it
under certain conditions; see file 'Copyright' for details.
Type :h for Help.  Top level.
...
```

sage.interfaces.lisp.reduce_load_Lisp()

EXAMPLES:

```
sage: from sage.interfaces.lisp import reduce_load_Lisp
sage: reduce_load_Lisp()
Lisp Interpreter
```

INTERFACE TO MACAULAY2

Note: You must have Macaulay2 installed on your computer for this interface to work. Macaulay2 is not included with Sage, but you can obtain it from <https://faculty.math.illinois.edu/Macaulay2/>. No additional optional Sage packages are required.

Sage provides an interface to the Macaulay2 computational algebra system. This system provides extensive functionality for commutative algebra. You do not have to install any optional packages.

The Macaulay2 interface offers three pieces of functionality:

- `macaulay2_console()` – A function that dumps you into an interactive command-line Macaulay2 session.
- `macaulay2.eval(expr)` – Evaluation of arbitrary Macaulay2 expressions, with the result returned as a string.
- `macaulay2(expr)` – Creation of a Sage object that wraps a Macaulay2 object. This provides a Pythonic interface to Macaulay2. For example, if `f = macaulay2(10)`, then `f.gcd(25)` returns the GCD of 10 and 25 computed using Macaulay2.

EXAMPLES:

```
sage: macaulay2('3/5 + 7/11')      # optional - macaulay2
68
--
55
sage: f = macaulay2('f = i -> i^3') # optional - macaulay2
sage: f                           # optional - macaulay2
f
sage: f(5)                         # optional - macaulay2
125

sage: R = macaulay2('ZZ/5[x,y,z]') # optional - macaulay2
sage: R                             # optional - macaulay2
ZZ
--[x...z]
5
sage: x = macaulay2('x')           # optional - macaulay2
sage: y = macaulay2('y')           # optional - macaulay2
sage: (x+y)^5                       # optional - macaulay2
5   5
x  + y
sage: parent((x+y)^5)              # optional - macaulay2
Macaulay2
```

The name of the variable to which a Macaulay2 element is assigned internally can be passed as an argument. This is useful for types like polynomial rings which acquire that name in Macaulay2:

```
sage: R = macaulay2('QQ[x,y,z,w]', 'R') # optional - macaulay2
sage: R # optional - macaulay2
R

sage: f = macaulay2('x^4 + 2*x*y^3 + x*y^2*w + x*y*z*w + x*y*w^2' # optional - macaulay2
↪macaulay2
.....: + 2*x*z*w^2 + y^4 + y^3*w + 2*y^2*z*w + z^4 + w^4')
sage: f # optional - macaulay2
4 3 4 4 2 3 2 2 2 4
x + 2x*y + y + z + x*y w + y w + x*y*z*w + 2y z*w + x*y*w + 2x*z*w + w
sage: g = f * macaulay2('x+y^5') # optional - macaulay2
sage: print(g.factor()) # optional - macaulay2
4 3 4 4 2 3 2 2 2 4 5
(x + 2x*y + y + z + x*y w + y w + x*y*z*w + 2y z*w + x*y*w + 2x*z*w + w)(y + x)
```

Use `eval()` for explicit control over what is sent to the interpreter. The argument is evaluated in Macaulay2 as is:

```
sage: macaulay2.eval('compactMatrixForm') # optional - macaulay2
true
sage: macaulay2.eval('compactMatrixForm = false;') # optional - macaulay2
sage: macaulay2.eval('matrix {{1, x^2+y}}') # optional - macaulay2
| 2 |
| 1 x + y |

1 2
Matrix R <--- R
sage: macaulay2.eval('compactMatrixForm = true;') # optional - macaulay2
```

AUTHORS:

- Kiran Kedlaya and David Roe (2006-02-05, during Sage coding sprint)
- William Stein (2006-02-09): inclusion in Sage; prompt uses regexp, calling of Macaulay2 functions via `__call__`.
- William Stein (2006-02-09): fixed bug in reading from file and improved output cleaning.
- Kiran Kedlaya (2006-02-12): added ring and ideal constructors, list delimiters, `is_Macaulay2Element`, `sage_polystring`, `__floordiv__`, `__mod__`, `__iter__`, `__len__`; stripped extra leading space and trailing newline from output.

Todo: Get rid of all numbers in output, e.g., in ideal function below.

```
class sage.interfaces.macaulay2.Macaulay2(maxread=None, script_subdirectory=None, logfile=None,
server=None, server_tmpdir=None, command=None)
Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.expect.Expect
```

Interface to the Macaulay2 interpreter.

clear(var)

Clear the variable named var.

The interface automatically clears Macaulay2 elements when they fall out of use, so calling this method is usually not necessary.

EXAMPLES:

```
sage: macaulay2.eval('R = QQ[x,y];') # optional - macaulay2
sage: macaulay2.eval('net class R') # optional - macaulay2
PolynomialRing
sage: macaulay2.clear('R')          # optional - macaulay2
sage: macaulay2.eval('net class R') # optional - macaulay2
Symbol
```

console()

Spawn a new M2 command-line session.

EXAMPLES:

```
sage: macaulay2.console() # not tested
Macaulay 2, version 1.1
with packages: Classic, Core, Elimination, IntegralClosure, LLBases, Parsing,
↳ PrimaryDecomposition, SchurRings, TangentCone
...
```

cputime(*t=None*)

EXAMPLES:

```
sage: R = macaulay2("QQ[x,y]") # optional - macaulay2
sage: x,y = R.gens()          # optional - macaulay2
sage: a = (x+y+1)^20         # optional - macaulay2
sage: macaulay2.cputime()    # optional - macaulay2; random
0.48393700000000001
```

eval(*code, strip=True, **kwds*)

Send the code *x* to the Macaulay2 interpreter and return the output as a string suitable for input back into Macaulay2, if possible.

INPUT:

- *code* – str
- *strip* – ignored

EXAMPLES:

```
sage: macaulay2.eval("2+2") # optional - macaulay2
4
```

get(*var*)

Get the value of the variable *var*.

INPUT:

- *var* - string; the name of the variable in Macaulay2

OUTPUT: a string of the textual representation of the variable in Macaulay2

EXAMPLES:

```
sage: macaulay2.set("a", "2") # optional - macaulay2
sage: macaulay2.get("a")     # optional - macaulay2
2
```

Note that the following syntax is used to obtain a Macaulay2Element instead:

```
sage: a = macaulay2('2'); a # optional - macaulay2
2
sage: type(a) # optional - macaulay2
<class 'sage.interfaces.macaulay2.Macaulay2Element'>
```

help(*s*)

EXAMPLES:

```
sage: macaulay2.help("load") # optional - macaulay2 - 1st call might be chatty.
↪...
...
sage: macaulay2.help("load") # optional - macaulay2
load...
****...
...
* "input" -- read Macaulay2 commands and echo
* "notify" -- whether to notify the user when a file is loaded...
```

ideal(**gens*)

Return the ideal generated by gens.

INPUT:

- gens – list or tuple of Macaulay2 objects (or objects that can be made into Macaulay2 objects via evaluation)

OUTPUT:

the Macaulay2 ideal generated by the given list of gens

EXAMPLES:

```
sage: R2 = macaulay2.ring('QQ', '[x, y]'); R2 # optional - macaulay2
QQ[x...y]
sage: I = macaulay2.ideal( ('y^2 - x^3', 'x - y') ); I # optional - macaulay2
      3      2
ideal (- x  + y , x - y)
sage: J = I^3; J.gb().gens().transpose() # optional - macaulay2
{-9} | y9-3y8+3y7-y6          |
{-7} | xy6-2xy5+xy4-y7+2y6-y5  |
{-5} | x2y3-x2y2-2xy4+2xy3+y5-y4 |
{-3} | x3-3x2y+3xy2-y3          |
```

new_from(*type*, *value*)

Return a new Macaulay2Element of type type constructed from value.

EXAMPLES:

```
sage: l = macaulay2.new_from("MutableList", [1,2,3]) # optional - macaulay2
sage: l # optional - macaulay2
MutableList{...3...}
sage: list(l) # optional - macaulay2
[1, 2, 3]
```

options(**get_value*, ***set_value*)

Global options for Macaulay2 elements.

OPTIONS:

- `after_print` – (default: `False`) append `AfterPrint` type information to textual representations

EXAMPLES:

```
sage: macaulay2.options.after_print = True # optional - macaulay2
sage: A = macaulay2(matrix([[1, 2], [3, 6]])); A # optional - macaulay2
| 1 2 |
| 3 6 |

      2      2
Matrix ZZ <--- ZZ
sage: A.kernel() # optional - macaulay2
image | 2 |
      | -1 |

      2
ZZ-module, submodule of ZZ
sage: macaulay2.options.after_print = False # optional - macaulay2
```

See [GlobalOptions](#) for more features of these options.

restart()

Restart Macaulay2 interpreter.

ring(`base_ring='ZZ', vars='[x]', order='Lex'`)

Create a Macaulay2 polynomial ring.

INPUT:

- `base_ring` – base ring (see examples below)
- `vars` – a tuple or string that defines the variable names
- `order` – string (default: ‘`Lex`’); the monomial order

OUTPUT: a Macaulay2 ring

EXAMPLES:

This is a ring in variables named `a` through `d` over the finite field of order 7, with graded reverse lex ordering:

```
sage: R1 = macaulay2.ring('ZZ/7', '[a..d]', 'GRevLex') # optional - macaulay2
sage: R1.describe() # optional - macaulay2
ZZ
--[a..d, Degrees => {4:1}, Heft => {1}, MonomialOrder => {MonomialSize => 16},
 7                                                    {GRevLex => {4:1} }
                                                    {Position => Up   }

-----
DegreeRank => 1]
sage: R1.char() # optional - macaulay2
7
```

This is a polynomial ring over the rational numbers:

```
sage: R2 = macaulay2.ring('QQ', '[x, y]') # optional - macaulay2
sage: R2.describe() # optional - macaulay2
QQ[x..y, Degrees => {2:1}, Heft => {1}, MonomialOrder => {MonomialSize => 16},
```

(continues on next page)

(continued from previous page)

```

{Lex => 2      }
{Position => Up  }
-----
DegreeRank => 1]

```

set(var, value)

Set the variable var to the given value.

INPUT:

- var - string; the name of the variable in Macaulay2
- value - a string to evaluate

EXAMPLES:

```

sage: macaulay2.set("a", "1+1") # optional - macaulay2
sage: macaulay2.get("a")       # optional - macaulay2
2

```

set_seed(seed=None)

Set the seed for Macaulay2 interpreter.

INPUT:

- seed – number (default: None). If None, it is set to a random number.

OUTPUT: the new seed

EXAMPLES:

```

sage: m = Macaulay2() # optional - macaulay2
sage: m.set_seed(123456) # optional - macaulay2
123456
sage: [m.random(100) for _ in range(11)] # optional - macaulay2
[8, 29, 5, 22, 4, 32, 35, 57, 3, 95, 36]

```

use(R)

Use the Macaulay2 ring R.

EXAMPLES:

```

sage: R = macaulay2("QQ[x,y]") # optional - macaulay2
sage: P = macaulay2("ZZ/7[symbol x, symbol y]") # optional - macaulay2
sage: macaulay2("x").cls()._operator('===', P) # optional - macaulay2
true
sage: macaulay2.use(R) # optional - macaulay2
sage: macaulay2("x").cls()._operator('===', R) # optional - macaulay2
true

```

version()

Returns the version of Macaulay2.

EXAMPLES:

```

sage: macaulay2.version() # optional - macaulay2
(1, 1...

```

```
class sage.interfaces.macaulay2.Macaulay2Element(parent, value, is_name=False, name=None)
Bases:   sage.interfaces.tab_completion.ExtraTabCompletion,  sage.interfaces.expect.
ExpectElement
```

Instances of this class represent objects in Macaulay2.

Using the method `sage()` we can translate some of them to SageMath objects:

`_sage_()`

EXAMPLES:

```
sage: macaulay2(ZZ).sage()          # optional - macaulay2, indirect doctest
Integer Ring
sage: macaulay2(QQ).sage()         # optional - macaulay2
Rational Field

sage: macaulay2(2).sage()          # optional - macaulay2
2
sage: macaulay2(1/2).sage()        # optional - macaulay2
1/2
sage: macaulay2(2/1).sage()        # optional - macaulay2
2
sage: _.parent()                   # optional - macaulay2
Rational Field
sage: macaulay2([1,2,3]).sage()    # optional - macaulay2
[1, 2, 3]

sage: m = matrix([[1,2],[3,4]])
sage: macaulay2(m).sage()          # optional - macaulay2
[1 2]
[3 4]

sage: D = macaulay2('hashTable {4 => 1, 2 => 3}') # optional - macaulay2
sage: D.pairs()                     # optional - macaulay2
{(4, 1), (2, 3)}
sage: D.sage() == {4: 1, 2: 3}      # optional - macaulay2
True

sage: macaulay2(QQ['x,y']).sage()   # optional - macaulay2
Multivariate Polynomial Ring in x, y over Rational Field
sage: macaulay2(QQ['x']).sage()     # optional - macaulay2
Univariate Polynomial Ring in x over Rational Field
sage: macaulay2(GF(7)['x,y']).sage() # optional - macaulay2
Multivariate Polynomial Ring in x, y over Finite Field of size 7

sage: macaulay2(GF(7)).sage()       # optional - macaulay2
Finite Field of size 7
sage: macaulay2(GF(49, 'a')).sage() # optional - macaulay2
Finite Field in a of size 7^2

sage: R.<x,y> = QQ[]
sage: macaulay2(x^2+y^2+1).sage()   # optional - macaulay2
x^2 + y^2 + 1

sage: R = macaulay2("QQ[x,y]")      # optional - macaulay2
```

(continues on next page)

(continued from previous page)

```

sage: I = macaulay2("ideal (x,y)")      # optional - macaulay2
sage: I.sage()                          # optional - macaulay2
Ideal (x, y) of Multivariate Polynomial Ring in x, y over Rational Field

sage: macaulay2("x = symbol x")        # optional - macaulay2
x
sage: macaulay2("QQ[x_0..x_25]").sage()  # optional - macaulay2
Multivariate Polynomial Ring in x_0, x_1,..., x_25 over Rational Field

sage: S = ZZ['x,y'].quotient('x^2-y')
sage: macaulay2(S).sage() == S          # optional - macaulay2
True
sage: S = GF(101)['x,y'].quotient('x^2-y')
sage: macaulay2(S).sage() == S          # optional - macaulay2
True

sage: R = GF(13)['a,b']['c,d']
sage: macaulay2(R).sage() == R          # optional - macaulay2
True
sage: macaulay2('a^2 + c').sage() == R('a^2 + c') # optional - macaulay2
True
sage: macaulay2.substitute('a', R).sage().parent() is R # optional - macaulay2
True

sage: R = macaulay2("QQ^2")            # optional - macaulay2
sage: R.sage()                          # optional - macaulay2
Vector space of dimension 2 over Rational Field

sage: macaulay2("vector {4_QQ, 2}").sage() # optional - macaulay2
(4, 2)
sage: _.parent()                          # optional - macaulay2
Vector space of dimension 2 over Rational Field

sage: m = macaulay2('"hello"')          # optional - macaulay2
sage: m.sage()                            # optional - macaulay2
'hello'

sage: gg = macaulay2.needsPackage('"Graphs"') # optional - macaulay2
sage: g = macaulay2.barbellGraph(3)      # optional - macaulay2
sage: g.sage()                            # optional - macaulay2
Graph on 6 vertices
sage: g.sage().edges(labels=False)        # optional - macaulay2
[(0, 1), (0, 2), (1, 2), (2, 3), (3, 4), (3, 5), (4, 5)]

sage: d = 'digraph ({1,2},{2,1},{3,1}), EntryMode => "edges"'
sage: g = macaulay2(d)                    # optional - macaulay2
sage: g.sage()                            # optional - macaulay2
Digraph on 3 vertices
sage: g.sage().edges(labels=False)        # optional - macaulay2
[(1, 2), (2, 1), (3, 1)]

```

Chain complexes and maps of chain complexes can be converted:

```

sage: R = ZZ['a,b,c']
sage: C = macaulay2(ideal(R.gens())).resolution() # optional - macaulay2
sage: ascii_art(C.sage()) # optional - macaulay2
    [-b  0 -c]    [ c]
    [ a -c  0]    [ a]
    [ 0  b  a]    [-b]
    [a b c]
0 <-- C_0 <----- C_1 <----- C_2 <----- C_3 <-- 0
sage: F = C.dot('dd') # optional - macaulay2
sage: G = F.sage() # optional - macaulay2
sage: G.in_degree(2) # optional - macaulay2
[-b  0 -c]
[ a -c  0]
[ 0  b  a]
sage: F.underscore(2).sage() == G.in_degree(2) # optional - macaulay2
True
sage: (F^2).sage() # optional - macaulay2
Chain complex morphism:
  From: Chain complex with at most 4 nonzero terms over Multivariate Polynomial_
↳Ring in a, b, c over Integer Ring
  To:   Chain complex with at most 4 nonzero terms over Multivariate Polynomial_
↳Ring in a, b, c over Integer Ring
    
```

Quotient rings in Macaulay2 inherit variable names from the ambient ring, so we mimic this behaviour in Sage:

```

sage: R = macaulay2("ZZ/7[x,y]") # optional - macaulay2
sage: I = macaulay2("ideal (x^3 - y^2)") # optional - macaulay2
sage: (R/I).gens() # optional - macaulay2
{x, y}
sage: (R/I).sage().gens() # optional - macaulay2
(x, y)
    
```

Elements of quotient rings:

```

sage: x, y = (R/I).gens() # optional - macaulay2
sage: f = ((x^3 + 2*y^2*x)^7).sage(); f # optional - macaulay2
2*x*y^18 + y^14
sage: f.parent() # optional - macaulay2
Quotient of Multivariate Polynomial Ring in x, y over Finite Field of size 7 by_
↳the ideal (x^3 - y^2)
    
```

after_print_text()

Obtain type information for this Macaulay2 element.

This is the text that is displayed using `AfterPrint` in a Macaulay2 interpreter.

Macaulay2 by default includes this information in the output. In Sage, this behavior can optionally be enabled by setting the option `after_print` in `Macaulay2.options`.

EXAMPLES:

```

sage: B = macaulay2(matrix([[1, 2], [3, 6]]).kernel()); B # optional -
↳macaulay2
image | 2 |
      | -1 |
    
```

(continues on next page)

(continued from previous page)

```
sage: B.after_print_text() # optional - macaulay2
                2
ZZ-module, submodule of ZZ
```

cls()

Since class is a keyword in Python, we have to use cls to call Macaulay2's class. In Macaulay2, class corresponds to Sage's notion of parent.

EXAMPLES:

```
sage: macaulay2(ZZ).cls() # optional - macaulay2
Ring
```

dot(x)

EXAMPLES:

```
sage: d = macaulay2.new("MutableHashTable") # optional - macaulay2
sage: d["k"] = 4                            # optional - macaulay2
sage: d.dot("k")                            # optional - macaulay2
4
```

external_string()

EXAMPLES:

```
sage: R = macaulay2("QQ[symbol x, symbol y]") # optional - macaulay2
sage: R.external_string()                   # optional - macaulay2
'QQ(monoid[x.y, Degrees => {2:1}, Heft => {1}, MonomialOrder => VerticalList
↪{MonomialSize => 32, GRevLex => {2:1}, Position => Up}, DegreeRank => 1)'
```

name(new_name=None)

Get or change the name of this Macaulay2 element.

INPUT:

- new_name – string (default: None). If None, return the name of this element; else return a new object identical to self whose name is new_name.

Note that this can overwrite existing variables in the system.

EXAMPLES:

```
sage: S = macaulay2(QQ['x,y']) # optional - macaulay2
sage: S.name()                 # optional - macaulay2
'sage...'
sage: R = S.name("R")          # optional - macaulay2
sage: R.name()                 # optional - macaulay2
'R'
sage: R.vars().cokernel().resolution() # optional - macaulay2
  1      2      1
R <-- R <-- R <-- 0
0      1      2      3
```

The name can also be given at definition:

```

sage: A = macaulay2(ZZ['x,y,z'], name='A') # optional - macaulay2
sage: A.name()                            # optional - macaulay2
'A'
sage: A^1                                  # optional - macaulay2
1
A

```

sage_polystring()

If this Macaulay2 element is a polynomial, return a string representation of this polynomial that is suitable for evaluation in Python. Thus `*` is used for multiplication and `**` for exponentiation. This function is primarily used internally.

EXAMPLES:

```

sage: R = macaulay2.ring('QQ', '(x,y)') # optional - macaulay2
sage: f = macaulay2('x^3 + 3*y^11 + 5') # optional - macaulay2
sage: print(f)                          # optional - macaulay2
 3    11
x  + 3y  + 5
sage: f.sage_polystring()                # optional - macaulay2
'x**3+3*y**11+5'

```

sharp(x)

EXAMPLES:

```

sage: a = macaulay2([1,2,3]) # optional - macaulay2
sage: a.sharp(0)            # optional - macaulay2
1

```

starstar(x)

The binary operator `**` in Macaulay2 is usually used for tensor or Cartesian power.

EXAMPLES:

```

sage: a = macaulay2([1,2]).set() # optional - macaulay2
sage: a.starstar(a)            # optional - macaulay2
set {(1, 1), (1, 2), (2, 1), (2, 2)}

```

structure_sheaf()

EXAMPLES:

```

sage: S = macaulay2('QQ[a..d]') # optional - macaulay2
sage: R = S / macaulay2('a^3 + b^3 + c^3 + d^3') # optional - macaulay2
sage: X = R.Proj().name('X') # optional - macaulay2
sage: X.structure_sheaf() # optional - macaulay2
doctest:...: DeprecationWarning: The function `structure_sheaf` is deprecated.
↳ Use `self.sheaf()` instead.
See https://trac.sagemath.org/27848 for details.
00
 X
sage: X.sheaf() # optional - macaulay2
00
 X

```

`subs(*args, **kws)`

Note that we have to override the substitute method so that we get the default one from Macaulay2 instead of the one provided by Element.

EXAMPLES:

```
sage: R = macaulay2("QQ[x]")          # optional - macaulay2
sage: P = macaulay2("ZZ/7[symbol x]") # optional - macaulay2
sage: x, = R.gens()                  # optional - macaulay2
sage: a = x^2 + 1                    # optional - macaulay2
sage: a = a.substitute(P)            # optional - macaulay2
sage: a.sage().parent()              # optional - macaulay2
Univariate Polynomial Ring in x over Finite Field of size 7
```

`substitute(*args, **kws)`

Note that we have to override the substitute method so that we get the default one from Macaulay2 instead of the one provided by Element.

EXAMPLES:

```
sage: R = macaulay2("QQ[x]")          # optional - macaulay2
sage: P = macaulay2("ZZ/7[symbol x]") # optional - macaulay2
sage: x, = R.gens()                  # optional - macaulay2
sage: a = x^2 + 1                    # optional - macaulay2
sage: a = a.substitute(P)            # optional - macaulay2
sage: a.sage().parent()              # optional - macaulay2
Univariate Polynomial Ring in x over Finite Field of size 7
```

`to_sage(*args, **kws)`

Deprecated: Use `sage()` instead. See [trac ticket #27848](#) for details.

`underscore(x)`

EXAMPLES:

```
sage: a = macaulay2([1,2,3]) # optional - macaulay2
sage: a.underscore(0)       # optional - macaulay2
1
```

`class sage.interfaces.macaulay2.Macaulay2Function(parent, name)`

Bases: `sage.interfaces.expect.ExpectFunction`

`class sage.interfaces.macaulay2.Macaulay2FunctionElement(obj, name)`

Bases: `sage.interfaces.expect.FunctionElement`

`sage.interfaces.macaulay2.is_Macaulay2Element(x)`

EXAMPLES:

```
sage: from sage.interfaces.macaulay2 import is_Macaulay2Element
sage: is_Macaulay2Element(2)          # optional - macaulay2
False
sage: is_Macaulay2Element(macaulay2(2)) # optional - macaulay2
True
```

`sage.interfaces.macaulay2.macaulay2_console()`

Spawn a new M2 command-line session.

EXAMPLES:

```
sage: macaulay2_console()                               # not tested
Macaulay 2, version 1.1
with packages: Classic, Core, Elimination, IntegralClosure, LLLBases, Parsing,
↳PrimaryDecomposition, SchurRings, TangentCone
...
```

`sage.interfaces.macaulay2.reduce_load_macaulay2()`

Used for reconstructing a copy of the Macaulay2 interpreter from a pickle.

EXAMPLES:

```
sage: from sage.interfaces.macaulay2 import reduce_load_macaulay2
sage: reduce_load_macaulay2()
Macaulay2
```

`sage.interfaces.macaulay2.remove_output_labels(s)`

Remove output labels of Macaulay2 from a string.

- s: output of Macaulay2
- s: string

Returns: the input string with n symbols removed from the beginning of each line, where n is the minimal number of spaces or symbols of Macaulay2 output labels (looking like 'o39 = ') present on every non-empty line.

Return type: string

Note: If s consists of several outputs and their labels have different width, it is possible that some strings will have leading spaces (or maybe even pieces of output labels). However, this function will try not cut any messages.

EXAMPLES:

```
sage: from sage.interfaces.macaulay2 import remove_output_labels
sage: output = 'o1 = QQ [x, y]\n\no1 : PolynomialRing\n'
sage: remove_output_labels(output)
'QQ [x, y]\n\nPolynomialRing\n'
```


INTERFACE TO MAGMA

Sage provides an interface to the Magma computational algebra system. This system provides extensive functionality for number theory, group theory, combinatorics and algebra.

Note: You must have Magma installed on your computer for this interface to work. Magma is not free, so it is not included with Sage, but you can obtain it from <http://magma.maths.usyd.edu.au/>.

The Magma interface offers three pieces of functionality:

1. `magma_console()` - A function that dumps you into an interactive command-line Magma session.
2. `magma.new(obj)` and alternatively `magma(obj)` - Creation of a Magma object from a Sage object `obj`. This provides a Pythonic interface to Magma. For example, if `f=magma.new(10)`, then `f.Factors()` returns the prime factorization of 10 computed using Magma. If `obj` is a string containing an arbitrary Magma expression, then the expression is evaluated in Magma to create a Magma object. An example is `magma.new('10 div 3')`, which returns Magma integer 3.
3. `magma.eval(expr)` - Evaluation of the Magma expression `expr`, with the result returned as a string.

Type `magma.[tab]` for a list of all functions available from your Magma. Type `magma.Function?` for Magma's help about the Magma Function.

22.1 Parameters

Some Magma functions have optional “parameters”, which are arguments that in Magma go after a colon. In Sage, you pass these using named function arguments. For example,

```
sage: E = magma('EllipticCurve([0,1,1,-1,0])')      # optional - magma
sage: E.Rank(Bound = 5)                          # optional - magma
0
```

22.2 Multiple Return Values

Some Magma functions return more than one value. You can control how many you get using the `nvals` named parameter to a function call:

```

sage: n = magma(100)                                # optional - magma
sage: n.IsSquare(nvals = 1)                          # optional - magma
true
sage: n.IsSquare(nvals = 2)                          # optional - magma
(true, 10)
sage: n = magma(-2006)                               # optional - magma
sage: n.Factorization()                             # optional - magma
[ <2, 1>, <17, 1>, <59, 1> ]
sage: n.Factorization(nvals=2)                       # optional - magma
([ <2, 1>, <17, 1>, <59, 1> ], -1)
    
```

We verify that an obviously principal ideal is principal:

```

sage: _ = magma.eval('R<x> := PolynomialRing(RationalField())') # optional - magma
sage: O = magma.NumberField('x^2+23').MaximalOrder()           # optional - magma
sage: I = magma('ideal<%s|%s.1>'%(O.name(),O.name()))         # optional - magma
sage: I.IsPrincipal(nvals=2)                                   # optional - magma
(true, [1, 0])
    
```

22.3 Long Input

The Magma interface reads in even very long input (using files) in a robust manner.

```

sage: t = '"%s"'%10^10000 # ten thousand character string.     # optional - magma
sage: a = magma.eval(t)                                       # optional - magma
sage: a = magma(t)                                           # optional - magma
    
```

22.4 Garbage Collection

There is a subtle point with the Magma interface, which arises from how garbage collection works. Consider the following session:

First, create a matrix `m` in Sage:

```

sage: m=matrix(ZZ,2,[1,2,3,4])                               # optional - magma
    
```

Then I create a corresponding matrix `A` in Magma:

```

sage: A = magma(m)                                          # optional - magma
    
```

It is called `_sage_...` in Magma:

```

sage: s = A.name(); s                                       # optional - magma
'_sage_...'
    
```

It's there:

```
sage: magma.eval(s) # optional - magma
'[1 2]\n[3 4]'
```

Now I delete the reference to that matrix:

```
sage: del A # optional - magma
```

Now `_sage_ [...]` is “zeroed out” in the Magma session:

```
sage: magma.eval(s) # optional - magma
'0'
```

If Sage did not do this garbage collection, then every single time you ever create any magma object from a sage object, e.g., by doing `magma(m)`, you would use up a lot of memory in that Magma session. This would lead to a horrible memory leak situation, which would make the Magma interface nearly useless for serious work.

22.5 Other Examples

We compute a space of modular forms with character.

```
sage: N = 20
sage: D = 20
sage: eps_top = fundamental_discriminant(D)
sage: eps = magma.KroneckerCharacter(eps_top, RationalField()) # optional - magma
sage: M2 = magma.ModularForms(eps) # optional - magma
sage: print(M2) # optional - magma
Space of modular forms on Gamma_1(5) ...
sage: print(M2.Basis()) # optional - magma
[
  1 + 10*q^2 + 20*q^3 + 20*q^5 + 60*q^7 + ...
  q + q^2 + 2*q^3 + 3*q^4 + 5*q^5 + 2*q^6 + ...
]
```

In Sage/Python (and sort of C++) coercion of an element x into a structure S is denoted by $S(x)$. This also works for the Magma interface:

```
sage: G = magma.DirichletGroup(20) # optional - magma
sage: G.AssignNames(['a', 'b']) # optional - magma
sage: (G.1).Modulus() # optional - magma
20
sage: e = magma.DirichletGroup(40)(G.1) # optional - magma
sage: print(e) # optional - magma
Kronecker character -4 in modulus 40
sage: print(e.Modulus()) # optional - magma
40
```

We coerce some polynomial rings into Magma:

```
sage: R.<y> = PolynomialRing(QQ)
sage: S = magma(R) # optional - magma
sage: print(S) # optional - magma
Univariate Polynomial Ring in y over Rational Field
```

(continues on next page)

(continued from previous page)

```
sage: S.1                                     # optional - magma
y
```

This example illustrates that Sage doesn't magically extend how Magma implicit coercion (what there is, at least) works. The errors below are the result of Magma having a rather limited automatic coercion system compared to Sage's:

```
sage: R.<x> = ZZ[]
sage: x * 5
5*x
sage: x * 1.0
x
sage: x * (2/3)
2/3*x
sage: y = magma(x)                           # optional - magma
sage: y * 5                                   # optional - magma
5*x
sage: y * 1.0                                 # optional - magma
$.1
sage: y * (2/3)                               # optional - magma
Traceback (most recent call last):
...
TypeError: Error evaluating Magma code.
...
Argument types given: RngUPolElt[RngInt], FldRatElt
```

AUTHORS:

- William Stein (2005): initial version
- William Stein (2006-02-28): added extensive tab completion and interactive IPython documentation support.
- William Stein (2006-03-09): added nvals argument for magma.functions...

```
class sage.interfaces.magma.Magma(script_subdirectory=None, logfile=None, server=None,
                                  server_tmppdir=None, user_config=False, seed=None, command=None)
```

```
  Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.expect.Expect
```

Interface to the Magma interpreter.

Type `magma.[tab]` for a list of all the functions available from your Magma install. Type `magma.Function?` for Magma's help about a given Function Type `magma(...)` to create a new Magma object, and `magma.eval(...)` to run a string using Magma (and get the result back as a string).

Note: If you do not own a local copy of Magma, try using the `magma_free` command instead, which uses the free demo web interface to Magma.

If you have ssh access to a remote installation of Magma, you can also set the `server` parameter to use it.

EXAMPLES:

You must use `nvals = 0` to call a function that doesn't return anything, otherwise you'll get an error. (`nvals` is the number of return values.)


```
sage: magma.set_nthreads(2)           #optional - magma
sage: magma.get_nthreads()           #optional - magma
2
```

GetVerbose(*type*)

Get the verbosity level of a given algorithm class etc. in Magma.

INPUT:

- *type* - string (e.g. 'Groebner'), see Magma documentation

EXAMPLES:

```
sage: magma.set_verbose("Groebner", 2)       # optional - magma
sage: magma.get_verbose("Groebner")         # optional - magma
2
```

SetNthreads(*n*)

Set the number of threads used for parallelized algorithms in Magma.

INPUT:

- *n* - number of threads

EXAMPLES:

```
sage: magma.set_nthreads(2)           #optional - magma
sage: magma.get_nthreads()           #optional - magma
2
```

SetVerbose(*type, level*)

Set the verbosity level for a given algorithm, class, etc. in Magma.

INPUT:

- *type* - string (e.g. 'Groebner')
- *level* - integer ≥ 0

EXAMPLES:

```
sage: magma.set_verbose("Groebner", 2)       # optional - magma
sage: magma.get_verbose("Groebner")         # optional - magma
2
```

attach(*filename*)

Attach the given file to the running instance of Magma.

Attaching a file in Magma makes all intrinsics defined in the file available to the shell. Moreover, if the file doesn't start with the `freeze;` command, then the file is reloaded whenever it is changed. Note that functions and procedures defined in the file are *not* available. For only those, use `magma.load(filename)`.

INPUT:

- *filename* - a string

EXAMPLES: Attaching a file that exists is fine:

```
sage: SAGE_EXTCODE = SAGE_ENV['SAGE_EXTCODE'] # optional - magma
sage: magma.attach('%s/magma/sage/basic.m'%SAGE_EXTCODE) # optional - magma
```

Attaching a file that doesn't exist raises an exception:

```

sage: SAGE_EXTCODE = SAGE_ENV['SAGE_EXTCODE']           # optional - magma
sage: magma.attach('%s/magma/sage/basic2.m'%SAGE_EXTCODE) # optional - magma
Traceback (most recent call last):
...
RuntimeError: Error evaluating Magma code...

```

attach_spec(filename)

Attach the given spec file to the running instance of Magma.

This can attach numerous other files to the running Magma (see the Magma documentation for more details).

INPUT:

- filename - a string

EXAMPLES:

```

sage: SAGE_EXTCODE = SAGE_ENV['SAGE_EXTCODE']           # optional - magma
sage: magma.attach_spec('%s/magma/spec'%SAGE_EXTCODE)   # optional - magma
sage: magma.attach_spec('%s/magma/spec2'%SAGE_EXTCODE) # optional - magma
Traceback (most recent call last):
...
RuntimeError: Can't open package spec file ../magma/spec2 for reading (No such
↪file or directory)

```

bar_call(left, name, gens, nvals=1)

This is a wrapper around the Magma constructor

nameleft gens

returning nvals.

INPUT:

- left - something coerceable to a magma object
- name - name of the constructor, e.g., sub, quo, ideal, etc.
- gens - if a list/tuple, each item is coerced to magma; otherwise gens itself is converted to magma
- nvals - positive integer; number of return values

OUTPUT: a single magma object if nvals == 1; otherwise a tuple of nvals magma objects.

EXAMPLES: The bar_call function is used by the sub, quo, and ideal methods of Magma elements. Here we illustrate directly using bar_call to create quotients:

```

sage: V = magma.RModule(ZZ,3)           # optional - magma
sage: V                                # optional - magma
RModule(IntegerRing(), 3)
sage: magma.bar_call(V, 'quo', [[1,2,3]], nvals=1) # optional - magma
RModule(IntegerRing(), 2)
sage: magma.bar_call(V, 'quo', [[1,2,3]], nvals=2) # optional - magma
(RModule(IntegerRing(), 2),
 Mapping from: RModule(IntegerRing(), 3) to RModule(IntegerRing(), 2))
sage: magma.bar_call(V, 'quo', V, nvals=2)         # optional - magma
(RModule(IntegerRing(), 0),
 Mapping from: RModule(IntegerRing(), 3) to RModule(IntegerRing(), 0))

```

chdir(*dir*)

Change the Magma interpreter's current working directory.

INPUT:

- *dir* – a string

EXAMPLES:

```
sage: magma.chdir('/')          # optional - magma
sage: magma.eval('System("pwd")')  # optional - magma
'/'
```

clear(*var*)

Clear the variable named *var* and make it available to be used again.

INPUT:

- *var* - a string

EXAMPLES:

```
sage: magma = Magma()           # optional - magma
sage: magma.clear('foo')         # sets foo to 0 in magma; optional - magma
sage: magma.eval('foo')         # optional - magma
'0'
```

Because we cleared *foo*, it is set to be used as a variable name in the future:

```
sage: a = magma('10')          # optional - magma
sage: a.name()                  # optional - magma
'foo'
```

The following tests that the whole variable clearing and freeing system is working correctly.

```
sage: magma = Magma()          # optional - magma
sage: a = magma('100')         # optional - magma
sage: a.name()                  # optional - magma
'__sage_[1]'
```

```
sage: del a                     # optional - magma
sage: b = magma('257')         # optional - magma
sage: b.name()                  # optional - magma
'__sage_[1]'
```

```
sage: del b                     # optional - magma
sage: magma('__sage_[1]')      # optional - magma
0
```

console()

Run a command line Magma session. This session is completely separate from this Magma interface.

EXAMPLES:

```
sage: magma.console()           # not tested
Magma V2.14-9    Sat Oct 11 2008 06:36:41 on one    [Seed = 1157408761]
Type ? for help. Type <Ctrl>-D to quit.
>
Total time: 2.820 seconds, Total memory usage: 3.95MB
```

cputime(*t=None*)

Return the CPU time in seconds that has elapsed since this Magma session started. This is a floating point number, computed by Magma.

If *t* is given, then instead return the floating point time from when *t* seconds had elapsed. This is useful for computing elapsed times between two points in a running program.

INPUT:

- *t* - float (default: None); if not None, return cputime since *t*

OUTPUT:

- float - seconds

EXAMPLES:

```
sage: type(magma.cputime())           # optional - magma
<... 'float'>
sage: magma.cputime()                # random, optional - magma
1.9399999999999999
sage: t = magma.cputime()            # optional - magma
sage: magma.cputime(t)               # random, optional - magma
0.02
```

eval(*x, strip=True, **kws*)

Evaluate the given block *x* of code in Magma and return the output as a string.

INPUT:

- *x* - string of code
- *strip* - ignored

OUTPUT: string

EXAMPLES:

We evaluate a string that involves assigning to a variable and printing.

```
sage: magma.eval("a := 10; print 2+a;")   # optional - magma
'12'
```

We evaluate a large input line (note that no weird output appears and that this works quickly).

```
sage: magma.eval("a := %s; %(10^10000)") # optional - magma
''
```

Verify that trac ticket #9705 is fixed:

```
sage: nl=chr(10) # newline character
sage: magma.eval( # optional - magma
..... "_<x>:=PolynomialRing(Rationals());"+nl+
..... "repeat"+nl+
..... "  g:=3*b*x^4+18*c*x^3-6*b^2*x^2-6*b*c*x-b^3-9*c^2 where b:=Random([-10..
↳10]) where c:=Random([-10..10]);"+nl+
..... "until g ne 0 and Roots(g) ne [];" +nl+
..... "print 'success';")
'success'
```

Verify that trac ticket #11401 is fixed:

```
sage: nl=chr(10) # newline character
sage: magma.eval("a:=3;" + nl + "b:=5;") == nl # optional - magma
True
sage: magma.eval("[a,b];") # optional - magma
'[ 3, 5 ]'
```

function_call(function, args=[], params={}, nvals=1)

Return result of evaluating a Magma function with given input, parameters, and asking for nvals as output.

INPUT:

- function - string, a Magma function name
- args - list of objects coercible into this magma interface
- params - Magma parameters, passed in after a colon
- nvals - number of return values from the function to ask Magma for

OUTPUT: MagmaElement or tuple of nvals MagmaElement's

EXAMPLES:

```
sage: magma.function_call('Factorization', 100) # optional - magma
[ <2, 2>, <5, 2> ]
sage: magma.function_call('NextPrime', 100, {'Proof':False}) # optional - magma
↪ magma
101
sage: magma.function_call('PolynomialRing', [QQ,2]) # optional - magma
Polynomial ring of rank 2 over Rational Field
Order: Lexicographical
Variables: $.1, $.2
```

Next, we illustrate multiple return values:

```
sage: magma.function_call('IsSquare', 100) # optional - magma
true
sage: magma.function_call('IsSquare', 100, nvals=2) # optional - magma
(true, 10)
sage: magma.function_call('IsSquare', 100, nvals=3) # optional - magma
Traceback (most recent call last):
...
RuntimeError: Error evaluating Magma code...
Runtime error in :=: Expected to assign 3 value(s) but only computed 2 value(s)
```

get(var)

Get the value of the variable var.

INPUT:

- var - string; name of a variable defined in the Magma session

OUTPUT:

- string - string representation of the value of the variable.

EXAMPLES:

```
sage: magma.set('abc', '2 + 3/5')      # optional - magma
sage: magma.get('abc')                 # optional - magma
'13/5'
```

get_nthreads()

Get the number of threads used in Magma.

EXAMPLES:

```
sage: magma.set_nthreads(2)           #optional - magma
sage: magma.get_nthreads()           #optional - magma
2
```

get_verbose(*type*)

Get the verbosity level of a given algorithm class etc. in Magma.

INPUT:

- *type* - string (e.g. 'Groebner'), see Magma documentation

EXAMPLES:

```
sage: magma.set_verbosity("Groebner", 2)      # optional - magma
sage: magma.get_verbosity("Groebner")        # optional - magma
2
```

help(*s*)

Return Magma help on string *s*.

This returns what typing ?*s* would return in Magma.

INPUT:

- *s* - string

OUTPUT: string

EXAMPLES:

```
sage: magma.help("NextPrime")           # optional - magma
=====
PATH: /magma/ring-field-algebra/integer/prime/next-previous/NextPrime
KIND: Intrinsic
=====
NextPrime(n) : RngIntElt -> RngIntElt
NextPrime(n: parameter) : RngIntElt -> RngIntElt
...
```

ideal(*L*)

Return the Magma ideal defined by *L*.

INPUT:

- *L* - a list of elements of a Sage multivariate polynomial ring.

OUTPUT: The magma ideal generated by the elements of *L*.

EXAMPLES:

```

sage: R.<x,y> = QQ[]
sage: magma.ideal([x^2, y^3*x])          # optional - magma
Ideal of Polynomial ring of rank 2 over Rational Field
Order: Graded Reverse Lexicographical
Variables: x, y
Homogeneous
Basis:
[
x^2,
x*y^3
]

```

load(filename)

Load the file with given filename using the 'load' command in the Magma shell.

Loading a file in Magma makes all the functions and procedures in the file available. The file should not contain any intrinsics (or you'll get errors). It also runs code in the file, which can produce output.

INPUT:

- filename - string

OUTPUT: output printed when loading the file

EXAMPLES:

```

sage: from tempfile import NamedTemporaryFile as NTF
sage: with NTF(mode="w+t", suffix=".m") as f: # optional - magma
....:     _ = f.write('function f(n) return n^2; end function;\nprint "hi";')
....:     print(magma.load(f.name))
Loading ".../a.m"
hi
sage: magma('f(12)') # optional - magma
144

```

objgens(value, gens)

Create a new object with given value and gens.

INPUT:

- value - something coercible to an element of this Magma interface
- gens - string; comma separated list of variable names

OUTPUT: new Magma element that is equal to value with given gens

EXAMPLES:

```

sage: R = magma.objgens('PolynomialRing(Rationals(),2)', 'alpha,beta') #
↪ optional - magma
sage: R.gens() # optional - magma
[alpha, beta]

```

Because of how Magma works you can use this to change the variable names of the generators of an object:

```

sage: S = magma.objgens(R, 'X,Y') # optional - magma
sage: R # optional - magma
Polynomial ring of rank 2 over Rational Field

```

(continues on next page)

(continued from previous page)

```
Order: Lexicographical
Variables: X, Y
sage: S # optional - magma
Polynomial ring of rank 2 over Rational Field
Order: Lexicographical
Variables: X, Y
```

set(var, value)

Set the variable var to the given value in the Magma interpreter.

INPUT:

- var - string; a variable name
- value - string; what to set var equal to

EXAMPLES:

```
sage: magma.set('abc', '2 + 3/5') # optional - magma
sage: magma('abc') # optional - magma
13/5
```

set_nthreads(n)

Set the number of threads used for parallelized algorithms in Magma.

INPUT:

- n - number of threads

EXAMPLES:

```
sage: magma.set_nthreads(2) #optional - magma
sage: magma.get_nthreads() #optional - magma
2
```

set_seed(seed=None)

Set the seed for the Magma interpreter.

The seed should be an integer.

EXAMPLES:

```
sage: m = Magma() # optional - magma
sage: m.set_seed(1) # optional - magma
1
sage: [m.Random(100) for i in range(5)] # optional - magma
[14, 81, 45, 75, 67]
```

set_verbose(type, level)

Set the verbosity level for a given algorithm, class, etc. in Magma.

INPUT:

- type – string (e.g. ‘Groebner’)
- level – integer ≥ 0

EXAMPLES:

```
sage: magma.set_verbose("Groebner", 2)      # optional - magma
sage: magma.get_verbose("Groebner")       # optional - magma
2
```

version()

Return the version of Magma that you have in your PATH on your computer.

OUTPUT:

- numbers - 3-tuple: major, minor, etc.
- string - version as a string

EXAMPLES:

```
sage: magma.version()           # random, optional - magma
((2, 14, 9), 'V2.14-9')
```

class `sage.interfaces.magma.MagmaElement`(parent, value, is_name=False, name=None)

Bases: `sage.interfaces.tab_completion.ExtraTabCompletion`, `sage.interfaces.expect.ExpectElement`

AssignNames(names)

EXAMPLES:

```
sage: S = magma.PolynomialRing(magma.Integers(), 2) # optional - magma
sage: S.AssignNames(['a', 'b'])                 # optional - magma
sage: S.1                                       # optional - magma
a
sage: S.1^2 + S.2 # optional - magma
a^2 + b
```

assign_names(names)

EXAMPLES:

```
sage: S = magma.PolynomialRing(magma.Integers(), 2) # optional - magma
sage: S.AssignNames(['a', 'b'])                 # optional - magma
sage: S.1                                       # optional - magma
a
sage: S.1^2 + S.2 # optional - magma
a^2 + b
```

eval(*args)

Evaluate self at the inputs.

INPUT:

- *args – import arguments

OUTPUT: self(*args)

EXAMPLES:

```
sage: f = magma('Factorization')             # optional - magma
sage: f.evaluate(15)                        # optional - magma
[ <3, 1>, <5, 1> ]
sage: f(15)                                 # optional - magma
[ <3, 1>, <5, 1> ]
```

(continues on next page)

(continued from previous page)

```

sage: f = magma('GCD') # optional - magma
sage: f.evaluate(15,20) # optional - magma
5
sage: m = matrix(QQ, 2, 2, [2,3,5,7]) # optional - magma
sage: f = magma('ElementaryDivisors') # optional - magma
sage: f.evaluate(m) # optional - magma
[ 1, 1 ]

```

evaluate(*args)

Evaluate self at the inputs.

INPUT:

- *args – import arguments

OUTPUT: self(*args)

EXAMPLES:

```

sage: f = magma('Factorization') # optional - magma
sage: f.evaluate(15) # optional - magma
[ <3, 1>, <5, 1> ]
sage: f(15) # optional - magma
[ <3, 1>, <5, 1> ]
sage: f = magma('GCD') # optional - magma
sage: f.evaluate(15,20) # optional - magma
5
sage: m = matrix(QQ, 2, 2, [2,3,5,7]) # optional - magma
sage: f = magma('ElementaryDivisors') # optional - magma
sage: f.evaluate(m) # optional - magma
[ 1, 1 ]

```

gen(*n*)

Return the *n*-th generator of this Magma element. Note that generators are 1-based in Magma rather than 0 based!

INPUT:

- *n* - a *positive* integer

OUTPUT: MagmaElement

EXAMPLES:

```

sage: k.<a> = GF(9)
sage: magma(k).gen(1) # optional -- magma
a
sage: R.<s,t,w> = k[]
sage: m = magma(R) # optional -- magma
sage: m.gen(1) # optional -- magma
s
sage: m.gen(2) # optional -- magma
t
sage: m.gen(3) # optional -- magma

```

(continues on next page)

(continued from previous page)

```

w
sage: m.gen(0)           # optional -- magma
Traceback (most recent call last):
...
IndexError: index must be positive since Magma indexes are 1-based
sage: m.gen(4)         # optional -- magma
Traceback (most recent call last):
...
IndexError: list index out of range

```

gen_names()

Return list of Magma variable names of the generators of self.

Note: As illustrated below, these are not the print names of the the generators of the Magma object, but special variable names in the Magma session that reference the generators.

EXAMPLES:

```

sage: R.<x,zw> = QQ[]
sage: S = magma(R)           # optional - magma
sage: S.gen_names()         # optional - magma
('_sage_[...]', '_sage_[...]')
sage: magma(S.gen_names()[1]) # optional - magma
zw

```

gens()

Return generators for self.

If self is named X in Magma, this function evaluates X.1, X.2, etc., in Magma until an error occurs. It then returns a Sage list of the resulting X.i. Note - I don't think there is a Magma command that returns the list of valid X.i. There are numerous ad hoc functions for various classes but nothing systematic. This function gets around that problem. Again, this is something that should probably be reported to the Magma group and fixed there.

AUTHORS:

- William Stein (2006-07-02)

EXAMPLES:

```

sage: magma("VectorSpace(RationalField(),3)").gens() # optional - magma
[(1 0 0), (0 1 0), (0 0 1)]
sage: magma("AbelianGroup(EllipticCurve([1..5]))").gens() # optional - magma
[$.1]

```

get_magma_attribute(attrname)

Return value of a given Magma attribute. This is like self.attrname in Magma.

OUTPUT: MagmaElement

EXAMPLES:

```

sage: V = magma("VectorSpace(RationalField(),10)") # optional - magma
sage: V.set_magma_attribute('M','hello')         # optional - magma

```

(continues on next page)

(continued from previous page)

```

sage: V.get_magma_attribute('M')             # optional - magma
hello
sage: V.M                                   # optional - magma
hello
    
```

ideal(gens)

Return the ideal of self with given list of generators.

INPUT:

- gens - object or list/tuple of generators

OUTPUT:

- magma element - a Magma ideal

EXAMPLES:

```

sage: R = magma('PolynomialRing(RationalField())') # optional - magma
sage: R.assign_names(['x'])                       # optional - magma
sage: x = R.1                                     # optional - magma
sage: R.ideal([x^2 - 1, x^3 - 1])                 # optional - magma
Ideal of Univariate Polynomial Ring in x over Rational Field generated by x - 1
    
```

list_attributes()

Return the attributes of self, obtained by calling the ListAttributes function in Magma.

OUTPUT: list of strings

EXAMPLES: We observe that vector spaces in Magma have numerous funny and mysterious attributes.

```

sage: V = magma("VectorSpace(RationalField(),2)") # optional - magma
sage: v = V.list_attributes(); v.sort()          # optional - magma
sage: print(v) # optional - magma
['Coroots', 'Involution', ..., 'p', 'ssbasis', 'weights']
    
```

methods(any=False)

Return signatures of all Magma intrinsics that can take self as the first argument, as strings.

INPUT:

- any - (bool: default is False) if True, also include signatures with Any as first argument.

OUTPUT: list of strings

EXAMPLES:

```

sage: v = magma('2/3').methods()                # optional - magma
sage: v[0]                                       # optional - magma
"!*'..."
    
```

quo(gens, **args)

Return the quotient of self by the given object or list of generators.

INPUT:

- gens - object or list/tuple of generators
- further named arguments that are ignored

OUTPUT:

- magma element - the quotient object
- magma element - mapping from self to the quotient object

EXAMPLES:

```
sage: V = magma('VectorSpace(RationalField(),3)') # optional - magma
sage: V.quo([[1,2,3], [1,1,2]]) # optional - magma
(Full Vector space of degree 1 over Rational Field, Mapping from: Full Vector
↳space of degree 3 over Rational Field to Full Vector space of degree 1 over
↳Rational Field)
```

We illustrate quotienting out by an object instead of a list of generators:

```
sage: W = V.sub([ [1,2,3], [1,1,2] ]) # optional - magma
sage: V.quo(W) # optional - magma
(Full Vector space of degree 1 over Rational Field, Mapping from: Full Vector
↳space of degree 3 over Rational Field to Full Vector space of degree 1 over
↳Rational Field)
```

We quotient a ZZ module out by a submodule.

```
sage: V = magma.RModule(ZZ,3); V # optional - magma
RModule(IntegerRing(), 3)
sage: W, phi = V.quo([[1,2,3]]) # optional - magma
sage: W # optional - magma
RModule(IntegerRing(), 2)
sage: phi # optional - magma
Mapping from: RModule(IntegerRing(), 3) to RModule(IntegerRing(), 2)
```

set_magma_attribute(attrname, value)

INPUT: attrname - string value - something coercible to a MagmaElement

EXAMPLES:

```
sage: V = magma("VectorSpace(RationalField(),2)") # optional - magma
sage: V.set_magma_attribute('M',10) # optional - magma
sage: V.get_magma_attribute('M') # optional - magma
10
sage: V.M # optional - magma
10
```

sub(gens)

Return the sub-object of self with given gens.

INPUT:

- gens - object or list/tuple of generators

EXAMPLES:

```
sage: V = magma('VectorSpace(RationalField(),3)') # optional - magma
sage: W = V.sub([ [1,2,3], [1,1,2] ]); W # optional - magma
Vector space of degree 3, dimension 2 over Rational Field
Generators:
(1 2 3)
(1 1 2)
```

(continues on next page)

(continued from previous page)

```
Echelonized basis:
(1 0 1)
(0 1 1)
```

class `sage.interfaces.magma.MagmaFunction`(*parent, name*)
 Bases: `sage.interfaces.expect.ExpectFunction`

class `sage.interfaces.magma.MagmaFunctionElement`(*obj, name*)
 Bases: `sage.interfaces.expect.FunctionElement`

class `sage.interfaces.magma.MagmaGBDefaultContext`(*magma=None*)
 Bases: `object`

Context to force preservation of verbosity options for Magma's Groebner basis computation.

class `sage.interfaces.magma.MagmaGBLogPrettyPrinter`(*verbosity=1, style='magma'*)
 Bases: `object`

A device which filters Magma Groebner basis computation logs.

flush()
 EXAMPLES:

```
sage: from sage.interfaces.magma import MagmaGBLogPrettyPrinter
sage: logs = MagmaGBLogPrettyPrinter()
sage: logs.flush()
```

write(s)
 EXAMPLES:

```
sage: P.<x,y,z> = GF(32003) []
sage: I = sage.rings.ideal.Katsura(P)
sage: _ = I.groebner_basis('magma',prot=True) # indirect doctest, optional -
↪magma
*****
FAUGERE F4 ALGORITHM
*****
...
Total Faugere F4 time: ..., real time: ...
```

`sage.interfaces.magma.extcode_dir`(*iface=None*)
 Return directory that contains all the Magma extcode. This is put in a writable directory owned by the user, since when attached, Magma has to write sig and lck files.

EXAMPLES:

```
sage: sage.interfaces.magma.extcode_dir()
'...dir.../data/'
```

`sage.interfaces.magma.is_MagmaElement`(*x*)
 Return True if x is of type MagmaElement, and False otherwise.

INPUT:

- x - any object

OUTPUT: bool

EXAMPLES:

```
sage: from sage.interfaces.magma import is_MagmaElement
sage: is_MagmaElement(2)
False
sage: is_MagmaElement(magma(2))           # optional - magma
True
```

`sage.interfaces.magma.magma_console()`

Run a command line Magma session.

EXAMPLES:

```
sage: magma_console()           # not tested
Magma V2.14-9   Sat Oct 11 2008 06:36:41 on one   [Seed = 1157408761]
Type ? for help.  Type <Ctrl>-D to quit.
>
Total time: 2.820 seconds, Total memory usage: 3.95MB
```

`sage.interfaces.magma.magma_gb_standard_options(func)`

Decorator to force default options for Magma.

EXAMPLES:

```
sage: P.<a,b,c,d,e> = PolynomialRing(GF(127))
sage: J = sage.rings.ideal.Cyclic(P).homogenize()
sage: from sage.misc.sageinspect import sage_getsource
sage: "mself" in sage_getsource(J._groebner_basis_magma)
True
```

`sage.interfaces.magma.reduce_load_Magma()`

Used in unpickling a Magma interface.

This functions just returns the global default Magma interface.

EXAMPLES:

```
sage: sage.interfaces.magma.reduce_load_Magma()
Magma
```

INTERFACE TO THE FREE ONLINE MAGMA CALCULATOR

class `sage.interfaces.magma_free.MagmaExpr`
Bases: `str`

class `sage.interfaces.magma_free.MagmaFree`
Bases: `object`

Evaluate MAGMA code without requiring that MAGMA be installed on your computer by using the free online MAGMA calculator.

EXAMPLES:

```
sage: magma_free("Factorization(9290348092384)") # optional - internet  
[ <2, 5>, <290323377887, 1> ]
```

eval(`x, **kws`)

`sage.interfaces.magma_free.magma_free_eval`(`code, strip=True, columns=0`)

Use the free online MAGMA calculator to evaluate the given input code and return the answer as a string.

LIMITATIONS: The code must evaluate in at most 20 seconds and there is a limitation on the amount of RAM.

EXAMPLES:

```
sage: magma_free("Factorization(9290348092384)") # optional - internet  
[ <2, 5>, <290323377887, 1> ]
```


INTERFACE TO MAPLE

AUTHORS:

- William Stein (2005): maple interface
- Gregg Musiker (2006-02-02): tutorial
- William Stein (2006-03-05): added tab completion, e.g., `maple.[tab]`, and help, e.g., `maple.sin?`.

You must have the optional commercial Maple interpreter installed and available as the command `maple` in your PATH in order to use this interface. You do not have to install any optional Sage packages.

Type `maple.[tab]` for a list of all the functions available from your Maple install. Type `maple.[tab]?` for Maple's help about a given function. Type `maple(...)` to create a new Maple object, and `maple.eval(...)` to run a string using Maple (and get the result back as a string).

EXAMPLES:

```
sage: maple('3 * 5')                                # optional - maple
15
sage: maple.eval('ifactor(2005)')                  # optional - maple
``(5)*``(401)'
sage: maple.ifactor(2005)                          # optional - maple
``(5)*``(401)
sage: maple.fsolve('x^2=cos(x)+4', 'x=0..5')       # optional - maple
1.914020619
sage: maple.factor('x^5 - y^5')                   # optional - maple
(x-y)*(x^4+x^3*y+x^2*y^2+x*y^3+y^4)
```

If the string “error” (case insensitive) occurs in the output of anything from Maple, a `RuntimeError` exception is raised.

24.1 Tutorial

AUTHORS:

- Gregg Musiker (2006-02-02): initial version.

This tutorial is based on the Maple Tutorial for number theory from <http://www.math.mun.ca/~drideout/m3370/numtheory.html>.

There are several ways to use the Maple Interface in Sage. We will discuss two of those ways in this tutorial.

1. If you have a maple expression such as

```
factor( (x^5-1) );
```

We can write that in sage as

```
sage: maple('factor(x^5-1)')          # optional - maple
(x-1)*(x^4+x^3+x^2+x+1)
```

Notice, there is no need to use a semicolon.

- Since Sage is written in Python, we can also import maple commands and write our scripts in a Pythonic way. For example, `factor()` is a maple command, so we can also factor in Sage using

```
sage: maple('(x^5-1)').factor()       # optional - maple
(x-1)*(x^4+x^3+x^2+x+1)
```

where `expression.command()` means the same thing as `command(expression)` in Maple. We will use this second type of syntax whenever possible, resorting to the first when needed.

```
sage: maple('(x^12-1)/(x-1)').simplify()  # optional - maple
(x+1)*(x^2+1)*(x^2+x+1)*(x^2-x+1)*(x^4-x^2+1)
```

The normal command will always reduce a rational function to the lowest terms. The `factor` command will factor a polynomial with rational coefficients into irreducible factors over the ring of integers. So for example,

```
sage: maple('(x^12-1)').factor()        # optional - maple
(x-1)*(x+1)*(x^2+x+1)*(x^2-x+1)*(x^2+1)*(x^4-x^2+1)
```

```
sage: maple('(x^28-1)').factor()       # optional - maple
(x-1)*(x^6+x^5+x^4+x^3+x^2+x+1)*(x+1)*(x^6-x^5+x^4-x^3+x^2-x+1)*(x^2+1)*(x^12-x^10+x^8-x^6+x^4-x^2+1)
```

Another important feature of maple is its online help. We can access this through sage as well. After reading the description of the command, you can press `q` to immediately get back to your original prompt.

Incidentally you can always get into a maple console by the command

```
sage: maple.console()                 # not tested
sage: !maple                           # not tested
```

Note that the above two commands are slightly different, and the first is preferred.

For example, for help on the maple command `fibonacci`, we type

```
sage: maple.help('fibonacci')         # not tested, since it uses a pager
```

We see there are two choices. Type

```
sage: maple.help('combinat, fibonacci') # not tested, since it uses a pager
```

We now see how the Maple command `fibonacci` works under the combinatorics package. Try typing in

```
sage: maple.fibonacci(10)             # optional - maple
fibonacci(10)
```

You will get `fibonacci(10)` as output since Maple has not loaded the combinatorics package yet. To rectify this type

```
sage: maple('combinat[fibonacci]')(10) # optional - maple
55
```

instead.

If you want to load the combinatorics package for future calculations, in Sage this can be done as

```
sage: maple.with_package('combinat')    # optional - maple
```

or

```
sage: maple.load('combinat')           # optional - maple
```

Now if we type `maple.fibonacci(10)`, we get the correct output:

```
sage: maple.fibonacci(10)              # optional - maple
55
```

Some common maple packages include `combinat`, `linalg`, and `numtheory`. To produce the first 19 Fibonacci numbers, use the sequence command.

```
sage: maple('seq(fibonacci(i),i=1..19)') # optional - maple
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584,
4181
```

Two other useful Maple commands are `ifactor` and `isprime`. For example

```
sage: maple.isprime(maple.fibonacci(27)) # optional - maple
false
sage: maple.ifactor(maple.fibonacci(27)) # optional - maple
``(2)*``(17)*``(53)*``(109)
```

Note that the `isprime` function that is included with Sage (which uses PARI) is better than the Maple one (it is faster and gives a provably correct answer, whereas Maple is sometimes wrong).

```
sage: alpha = maple('(1+sqrt(5))/2')      # optional - maple
sage: beta  = maple('(1-sqrt(5))/2')      # optional - maple
sage: f19   = alpha^19 - beta^19/maple('sqrt(5)') # optional - maple
sage: f19   # optional - maple
(1/2+1/2*5^(1/2))^19-1/5*(1/2-1/2*5^(1/2))^19*5^(1/2)
sage: f19.simplify()                     # somewhat randomly ordered output; optional - maple
6765+5778/5*5^(1/2)
```

Let's say we want to write a maple program now that squares a number if it is positive and cubes it if it is negative. In maple, that would look like

```
mysqcu := proc(x)
if x > 0 then x^2;
else x^3; fi;
end;
```

In Sage, we write

```
sage: mysqcu = maple('proc(x) if x > 0 then x^2 else x^3 fi end') # optional - maple
sage: mysqcu(5)                                               # optional - maple
25
sage: mysqcu(-5)                                             # optional - maple
-125
```

More complicated programs should be put in a separate file and loaded.

```
class sage.interfaces.maple.Maple(maxread=None, script_subdirectory=None, server=None,
                                  server_impdir=None, logfile=None, ulimit=None)
```

Bases: `sage.interfaces.tab_completion.ExtraTabCompletion`, `sage.interfaces.expect.Expect`

Interface to the Maple interpreter.

Type `maple.[tab]` for a list of all the functions available from your Maple install. Type `maple.[tab]?` for Maple’s help about a given function. Type `maple(...)` to create a new Maple object, and `maple.eval(...)` to run a string using Maple (and get the result back as a string).

```
clear(var)
```

Clear the variable named `var`.

To clear a Maple variable, you must assign ‘itself’ to itself. In Maple ‘expr’ prevents `expr` to be evaluated.

EXAMPLES:

```
sage: maple.set('xx', '2') # optional - maple
sage: maple.get('xx')     # optional - maple
'2'
sage: maple.clear('xx')  # optional - maple
sage: maple.get('xx')    # optional - maple
'xx'
```

```
completions(s)
```

Return all commands that complete the command starting with the string `s`. This is like typing `s[Ctrl-T]` in the maple interpreter.

EXAMPLES:

```
sage: c = maple.completions('di') # optional - maple
sage: 'divide' in c               # optional - maple
True
```

```
console()
```

Spawn a new Maple command-line session.

EXAMPLES:

```
sage: maple.console() # not tested
|^/|      Maple 11 (IBM INTEL LINUX)
_._|\|    |/_|_ . Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2007
 \ MAPLE / All rights reserved. Maple is a trademark of
 <____>   Waterloo Maple Inc.
   |      Type ? for help.
>
```

```
cputime(t=None)
```

Return the amount of CPU time that the Maple session has used. If `t` is not `None`, then it returns the difference between the current CPU time and `t`.

EXAMPLES:

```
sage: t = maple.cputime() # optional - maple
sage: t                  # random; optional - maple
0.02
```

(continues on next page)

(continued from previous page)

```

sage: x = maple('x')          # optional - maple
sage: maple.diff(x^2, x)      # optional - maple
2*x
sage: maple.cputime(t)        # random; optional - maple
0.0

```

expect()

Return the pexpect object for this Maple session.

EXAMPLES:

```

sage: m = Maple()             # optional - maple
sage: m.expect() is None     # optional - maple
True
sage: m._start()              # optional - maple
sage: m.expect()              # optional - maple
Maple with PID ...
sage: m.quit()                # optional - maple

```

get(*var*)Get the value of the variable *var*.

EXAMPLES:

```

sage: maple.set('xx', '2')   # optional - maple
sage: maple.get('xx')        # optional - maple
'2'

```

help(*string*)Display Maple help about *string*.

This is the same as typing “?string” in the Maple console.

INPUT:

- *string* - a string to search for in the maple help system

EXAMPLES:

```

sage: maple.help('Psi')      # not tested
Psi - the Digamma and Polygamma functions
...

```

load(*package*)

Make a package of Maple procedures available in the interpreter.

INPUT:

- *package* - string

EXAMPLES: Some functions are unknown to Maple until you use with to include the appropriate package.

```

sage: maple.quit()           # reset maple; optional -- maple
sage: maple('partition(10)') # optional - maple
partition(10)
sage: maple('bell(10)')      # optional - maple
bell(10)

```

(continues on next page)

(continued from previous page)

```

sage: maple.with_package('combinat')          # optional - maple
sage: maple('partition(10)')                  # optional - maple
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 2], [1, 1, 1, 1, 1, 1,
↪ 2, 2], [1, 1, 1, 1, 2, 2, 2], [1, 1, 2, 2, 2, 2], [2, 2, 2, 2, 2], [1, 1, 1,
↪ 1, 1, 1, 1, 3], [1, 1, 1, 1, 1, 2, 3], [1, 1, 1, 2, 2, 3], [1, 2, 2, 2, 3],
↪ 1, 1, 1, 1, 3, 3], [1, 1, 2, 3, 3], [2, 2, 3, 3], [1, 3, 3, 3], [1, 1, 1, 1,
↪ 1, 1, 4], [1, 1, 1, 1, 2, 4], [1, 1, 2, 2, 4], [2, 2, 2, 4], [1, 1, 1, 3, 4],
↪ 1, 2, 3, 4], [3, 3, 4], [1, 1, 4, 4], [2, 4, 4], [1, 1, 1, 1, 1, 5], [1, 1,
↪ 1, 2, 5], [1, 2, 2, 5], [1, 1, 3, 5], [2, 3, 5], [1, 4, 5], [5, 5], [1, 1, 1,
↪ 1, 6], [1, 1, 2, 6], [2, 2, 6], [1, 3, 6], [4, 6], [1, 1, 1, 7], [1, 2, 7],
↪ 3, 7], [1, 1, 8], [2, 8], [1, 9], [10]]
sage: maple('bell(10)')                       # optional - maple
115975
sage: maple('fibonacci(10)')                   # optional - maple
55

```

set(var, value)

Set the variable var to the given value.

EXAMPLES:

```

sage: maple.set('xx', '2') # optional - maple
sage: maple.get('xx')      # optional - maple
'2'

```

source(s)

Display the Maple source (if possible) about s. This is the same as returning the output produced by the following Maple commands:

```
interface(verboseproc=2): print(s)
```

INPUT:

- s - a string representing the function whose source code you want

EXAMPLES:

```

sage: maple.source('curry') #not tested
p -> subs('_X' = args[2 .. nargs], () -> p(_X, args))

```

with_package(package)

Make a package of Maple procedures available in the interpreter.

INPUT:

- package - string

EXAMPLES: Some functions are unknown to Maple until you use with to include the appropriate package.

```

sage: maple.quit() # reset maple; optional -- maple
sage: maple('partition(10)') # optional - maple
partition(10)
sage: maple('bell(10)')      # optional - maple
bell(10)
sage: maple.with_package('combinat') # optional - maple
sage: maple('partition(10)') # optional - maple

```

(continues on next page)

(continued from previous page)

```
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 2], [1, 1, 1, 1, 1, 1,
↪ 2, 2], [1, 1, 1, 1, 2, 2, 2], [1, 1, 2, 2, 2, 2], [2, 2, 2, 2, 2], [1, 1, 1,
↪ 1, 1, 1, 3], [1, 1, 1, 1, 1, 2, 3], [1, 1, 1, 2, 2, 3], [1, 2, 2, 2, 3],
↪ [1, 1, 1, 1, 3, 3], [1, 1, 2, 3, 3], [2, 2, 3, 3], [1, 3, 3, 3], [1, 1, 1, 1,
↪ 1, 1, 4], [1, 1, 1, 1, 2, 4], [1, 1, 2, 2, 4], [2, 2, 2, 4], [1, 1, 1, 3, 4],
↪ [1, 2, 3, 4], [3, 3, 4], [1, 1, 4, 4], [2, 4, 4], [1, 1, 1, 1, 1, 5], [1, 1,
↪ 1, 2, 5], [1, 2, 2, 5], [1, 1, 3, 5], [2, 3, 5], [1, 4, 5], [5, 5], [1, 1, 1,
↪ 1, 6], [1, 1, 2, 6], [2, 2, 6], [1, 3, 6], [4, 6], [1, 1, 1, 7], [1, 2, 7],
↪ [3, 7], [1, 1, 8], [2, 8], [1, 9], [10]]
sage: maple('bell(10)')              # optional - maple
115975
sage: maple('fibonacci(10)')        # optional - maple
55
```

class `sage.interfaces.maple.MapleElement`(*parent, value, is_name=False, name=None*)
 Bases: `sage.interfaces.tab_completion.ExtraTabCompletion`, `sage.interfaces.expect.ExpectElement`

op(*i=None*)
 Return the i-th operand of this expression.

INPUT:

- *i* – an integer or `None`

EXAMPLES:

```
sage: V = maple(vector(QQ,[4,5,6]))    # optional - maple
sage: V.op(1)                         # optional - maple
3
sage: V.op(2)                         # optional - maple
{1 = 4, 2 = 5, 3 = 6}
```

class `sage.interfaces.maple.MapleFunction`(*parent, name*)
 Bases: `sage.interfaces.expect.ExpectFunction`

class `sage.interfaces.maple.MapleFunctionElement`(*obj, name*)
 Bases: `sage.interfaces.expect.FunctionElement`

`sage.interfaces.maple.maple_console`()
 Spawn a new Maple command-line session.

EXAMPLES:

```
sage: maple_console() #not tested
|^/|    Maple 11 (IBM INTEL LINUX)
._|\|  |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2007
\ MAPLE / All rights reserved. Maple is a trademark of
<____ ____> Waterloo Maple Inc.
      |    Type ? for help.
>
```

`sage.interfaces.maple.reduce_load_Maple`()
 Return the maple object created in `sage.interfaces.maple`.

EXAMPLES:

```
sage: from sage.interfaces.maple import reduce_load_Maple
sage: reduce_load_Maple()
Maple
```

INTERFACE TO MATHEMATICA

The Mathematica interface will only work if Mathematica is installed on your computer with a command line interface that runs when you give the `math` command. The interface lets you send certain Sage objects to Mathematica, run Mathematica functions, import certain Mathematica expressions to Sage, or any combination of the above. The Sage command:

```
sage: print(mathematica._install_hints())
...
```

prints more information on Mathematica installation.

To send a Sage object `sobj` to Mathematica, call `mathematica(sobj)`. This exports the Sage object to Mathematica and returns a new Sage object wrapping the Mathematica expression/variable, so that you can use the Mathematica variable from within Sage. You can then call Mathematica functions on the new object; for example:

```
sage: mobj = mathematica(x^2-1)          # optional - mathematica
sage: mobj.Factor()                    # optional - mathematica
(-1 + x)*(1 + x)
```

In the above example the factorization is done using Mathematica's `Factor[]` function.

To see Mathematica's output you can simply print the Mathematica wrapper object. However if you want to import Mathematica's output back to Sage, call the Mathematica wrapper object's `sage()` method. This method returns a native Sage object:

```
sage: mobj = mathematica(x^2-1)          # optional - mathematica
sage: mobj2 = mobj.Factor(); mobj2      # optional - mathematica
(-1 + x)*(1 + x)
sage: mobj2.parent()                   # optional - mathematica
Mathematica
sage: sobj = mobj2.sage(); sobj         # optional - mathematica
(x + 1)*(x - 1)
sage: sobj.parent()                    # optional - mathematica
Symbolic Ring
```

If you want to run a Mathematica function and don't already have the input in the form of a Sage object, then it might be simpler to input a string to `mathematica(expr)`. This string will be evaluated as if you had typed it into Mathematica:

```
sage: mathematica('Factor[x^2-1]')      # optional - mathematica
(-1 + x)*(1 + x)
sage: mathematica('Range[3]')          # optional - mathematica
{1, 2, 3}
```

If you don't want Sage to go to the trouble of creating a wrapper for the Mathematica expression, then you can call `mathematica.eval(expr)`, which returns the result as a Mathematica AsciiArtString formatted string. If you want the result to be a string formatted like Mathematica's InputForm, call `repr(mobj)` on the wrapper object `mobj`. If you want a string formatted in Sage style, call `mobj._sage_repr()`:

```
sage: mathematica.eval('x^2 - 1')          # optional - mathematica
      2
      -1 + x
sage: repr(mathematica('Range[3]'))      # optional - mathematica
'{1, 2, 3}'
sage: mathematica('Range[3]')._sage_repr() # optional - mathematica
'[1, 2, 3]'
```

Finally, if you just want to use a Mathematica command line from within Sage, the function `mathematica_console()` dumps you into an interactive command-line Mathematica session. This is an enhanced version of the usual Mathematica command-line, in that it provides readline editing and history (the usual one doesn't!)

25.1 Tutorial

We follow some of the tutorial from <http://library.wolfram.com/conferences/devconf99/withoff/Basic1.html/>.

For any of this to work you must buy and install the Mathematica program, and it must be available as the command `math` in your `PATH`.

25.1.1 Syntax

Now make 1 and add it to itself. The result is a Mathematica object.

```
sage: m = mathematica
sage: a = m(1) + m(1); a          # optional - mathematica
      2
sage: a.parent()                 # optional - mathematica
Mathematica
sage: m('1+1')                   # optional - mathematica
      2
sage: m(3)**m(50)                 # optional - mathematica
717897987691852588770249
```

The following is equivalent to `Plus[2, 3]` in Mathematica:

```
sage: m = mathematica
sage: m(2).Plus(m(3))            # optional - mathematica
5
```

We can also compute $7(2 + 3)$.

```
sage: m(7).Times(m(2).Plus(m(3))) # optional - mathematica
35
sage: m('7(2+3)')                # optional - mathematica
35
```

25.1.2 Some typical input

We solve an equation and a system of two equations:

```
sage: eqn = mathematica('3x + 5 == 14') # optional - mathematica
sage: eqn                               # optional - mathematica
5 + 3*x == 14
sage: eqn.Solve('x')                   # optional - mathematica
{{x -> 3}}
```

```
sage: sys = mathematica('{x^2 - 3y == 3, 2x - y == 1}') # optional - mathematica
sage: print(sys)                        # optional - mathematica
      2
      {x  - 3 y == 3, 2 x - y == 1}
sage: sys.Solve('{x, y}')               # optional - mathematica
{{x -> 0, y -> -1}, {x -> 6, y -> 11}}
```

25.1.3 Assignments and definitions

If you assign the mathematica 5 to a variable *c* in Sage, this does not affect the *c* in Mathematica.

```
sage: c = m(5)                          # optional - mathematica
sage: print(m('b + c x'))               # optional - mathematica
      b + c x
sage: print(m('b') + c*m('x'))        # optional - mathematica
      b + 5 x
```

The Sage interfaces changes Sage lists into Mathematica lists:

```
sage: m = mathematica
sage: eq1 = m('x^2 - 3y == 3')           # optional - mathematica
sage: eq2 = m('2x - y == 1')           # optional - mathematica
sage: v = m([eq1, eq2]); v              # optional - mathematica
{x^2 - 3*y == 3, 2*x - y == 1}
sage: v.Solve(['x', 'y'])               # optional - mathematica
{{x -> 0, y -> -1}, {x -> 6, y -> 11}}
```

25.1.4 Function definitions

Define mathematica functions by simply sending the definition to the interpreter.

```
sage: m = mathematica
sage: _ = mathematica('f[p_] = p^2');   # optional - mathematica
sage: m('f[9]')                         # optional - mathematica
81
```

25.1.5 Numerical Calculations

We find the x such that $e^x - 3x = 0$.

```
sage: eqn = mathematica('Exp[x] - 3x == 0') # optional - mathematica
sage: eqn.FindRoot(['x', 2]) # optional - mathematica
{x -> 1.512134551657842}
```

Note that this agrees with what the PARI interpreter gp produces:

```
sage: gp('solve(x=1,2,exp(x)-3*x)')
1.512134551657842473896739678 # 32-bit
1.5121345516578424738967396780720387046 # 64-bit
```

Next we find the minimum of a polynomial using the two different ways of accessing Mathematica:

```
sage: mathematica('FindMinimum[x^3 - 6x^2 + 11x - 5, {x,3}]') # optional - mathematica
{0.6150998205402516, {x -> 2.5773502699629733}}
sage: f = mathematica('x^3 - 6x^2 + 11x - 5') # optional - mathematica
sage: f.FindMinimum(['x', 3]) # optional - mathematica
{0.6150998205402516, {x -> 2.5773502699629733}}
```

25.1.6 Polynomial and Integer Factorization

We factor a polynomial of degree 200 over the integers.

```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = (x**100+17*x+5)*(x**100-5*x+20)
sage: f
x^200 + 12*x^101 + 25*x^100 - 85*x^2 + 315*x + 100
sage: g = mathematica(str(f)) # optional - mathematica
sage: print(g) # optional - mathematica
      2      100      101      200
100 + 315 x - 85 x + 25 x + 12 x + x
sage: g # optional - mathematica
100 + 315*x - 85*x^2 + 25*x^100 + 12*x^101 + x^200
sage: print(g.Factor()) # optional - mathematica
      100      100
(20 - 5 x + x ) (5 + 17 x + x )
```

We can also factor a multivariate polynomial:

```
sage: f = mathematica('x^6 + (-y - 2)*x^5 + (y^3 + 2*y)*x^4 - y^4*x^3') # optional -
↪mathematica
sage: print(f.Factor()) # optional - mathematica
      3      2      3
x (x - y) (-2 x + x + y )
```

We factor an integer:

```
sage: n = mathematica(2434500) # optional - mathematica
sage: n.FactorInteger() # optional - mathematica
{{2, 2}, {3, 2}, {5, 3}, {541, 1}}
```

(continues on next page)

(continued from previous page)

```

sage: n = mathematica(2434500)           # optional - mathematica
sage: F = n.FactorInteger(); F          # optional - mathematica
{{2, 2}, {3, 2}, {5, 3}, {541, 1}}
sage: F[1]                               # optional - mathematica
{2, 2}
sage: F[4]                               # optional - mathematica
{541, 1}

```

Mathematica's ECM package is no longer available.

25.2 Long Input

The Mathematica interface reads in even very long input (using files) in a robust manner.

```

sage: t = "%s"%10^10000 # ten thousand character string.
sage: a = mathematica(t) # optional - mathematica
sage: a = mathematica.eval(t) # optional - mathematica

```

25.3 Loading and saving

Mathematica has an excellent `InputForm` function, which makes saving and loading Mathematica objects possible. The first examples test saving and loading to strings.

```

sage: x = mathematica(pi/2) # optional - mathematica
sage: print(x)             # optional - mathematica
Pi
--
2
sage: loads(dumps(x)) == x # optional - mathematica
True
sage: n = x.N(50)         # optional - mathematica
sage: print(n)           # optional - mathematica
1.5707963267948966192313216916397514420985846996876
sage: loads(dumps(n)) == n # optional - mathematica
True

```

25.4 Complicated translations

The `obj.sage()` method tries to convert a Mathematica object to a Sage object. In many cases, it will just work. In particular, it should be able to convert expressions entirely consisting of:

- numbers, i.e. integers, floats, complex numbers;
- functions and named constants also present in Sage, where:
 - Sage knows how to translate the function or constant's name from Mathematica's, or
 - the Sage name for the function or constant is trivially related to Mathematica's;
- symbolic variables whose names don't pathologically overlap with objects already defined in Sage.

This method will not work when Mathematica's output includes:

- strings;
- functions unknown to Sage;
- Mathematica functions with different parameters/parameter order to the Sage equivalent.

If you want to convert more complicated Mathematica expressions, you can instead call `mobj._sage_()` and supply a translation dictionary:

```
sage: m = mathematica('NewFn[x]')      # optional - mathematica
sage: m._sage_(locals={'NewFn', 1}: sin}) # optional - mathematica
sin(x)
```

For more details, see the documentation for `._sage_()`.

OTHER Examples:

```
sage: def math_bessel_K(nu, x):
.....:     return mathematica(nu).BesselK(x).N(20)
sage: math_bessel_K(2, I)              # optional - mathematica
-2.59288617549119697817 + 0.18048997206696202663*I
```

```
sage: slist = [[1, 2], 3., 4 + I]
sage: mlist = mathematica(slist); mlist # optional - mathematica
{{1, 2}, 3., 4 + I}
sage: slist2 = list(mlist); slist2      # optional - mathematica
[[1, 2], 3., 4 + I]
sage: slist2[0]                        # optional - mathematica
{1, 2}
sage: slist2[0].parent()                # optional - mathematica
Mathematica
sage: slist3 = mlist.sage(); slist3     # optional - mathematica
[[1, 2], 3.0000000000000000, I + 4]
```

```
sage: mathematica('10.^80')           # optional - mathematica
1.*^80
sage: mathematica('10.^80').sage()    # optional - mathematica
1.0000000000000000e80
```

AUTHORS:

- William Stein (2005): first version
- Doug Cutrell (2006-03-01): Instructions for use under Cygwin/Windows.
- Felix Lawrence (2009-08-21): Added support for importing Mathematica lists and floats with exponents.

```
class sage.interfaces.mathematica.Mathematica(maxread=None, script_subdirectory=None,
                                               logfile=None, server=None, server_tmpdir=None,
                                               command=None, verbose_start=False)
```

Bases: `sage.interfaces.tab_completion.ExtraTabCompletion`, `sage.interfaces.expect.Expect`

Interface to the Mathematica interpreter.

chdir(*dir*)

Change Mathematica's current working directory.

EXAMPLES:

```
sage: mathematica.chdir('/')           # optional - mathematica
sage: mathematica('Directory[]')     # optional - mathematica
"/"
```

console(*readline=True*)

eval(*code, strip=True, **kwds*)

get(*var, ascii_art=False*)

Get the value of the variable *var*.

AUTHORS:

- William Stein
- Kiran Kedlaya (2006-02-04): suggested using InputForm

help(*cmd*)

set(*var, value*)

Set the variable *var* to the given value.

class `sage.interfaces.mathematica.MathematicaElement`(*parent, value, is_name=False, name=None*)

Bases: `sage.interfaces.expect.ExpectElement`

n(**args, **kwargs*)

Numerical approximation by converting to Sage object first

Convert the object into a Sage object and return its numerical approximation. See documentation of the function `sage.misc.functional.n()` for details.

EXAMPLES:

```
sage: mathematica('Pi').n(10)         # optional -- mathematica
3.1
sage: mathematica('Pi').n()          # optional -- mathematica
3.14159265358979
sage: mathematica('Pi').n(digits=10) # optional -- mathematica
3.141592654
```

save_image(*filename, ImageSize=600*)

Save a mathematica graphics

INPUT:

- *filename* – string. The filename to save as. The extension determines the image file format.
- *ImageSize* – integer. The size of the resulting image.

EXAMPLES:

```
sage: P = mathematica('Plot[Sin[x],{x,-2Pi,4Pi}]') # optional - mathematica
sage: filename = tmp_filename()                  # optional - mathematica
sage: P.save_image(filename, ImageSize=800)      # optional -
↪mathematica
```

show(*ImageSize=600*)

Show a mathematica expression immediately.

This method attempts to display the graphics immediately, without waiting for the currently running code (if any) to return to the command line. Be careful, calling it from within a loop will potentially launch a large number of external viewer programs.

INPUT:

- `ImageSize` – integer. The size of the resulting image.

OUTPUT:

This method does not return anything. Use `save()` if you want to save the figure as an image.

EXAMPLES:

```
sage: Q = mathematica('Sin[x Cos[y]]/Sqrt[1-x^2]') # optional - mathematica
sage: show(Q) # optional - mathematica
Sin[x*Cos[y]]/Sqrt[1 - x^2]
```

The following example starts a Mathematica frontend to do the rendering (trac ticket #28819):

```
sage: P = mathematica('Plot[Sin[x], {x, -2Pi, 4Pi}]') # optional - mathematica
sage: show(P) # optional - mathematica
↳mathematicafrontend
sage: P.show(ImageSize=800) # optional - mathematica
↳mathematicafrontend
```

str()

class `sage.interfaces.mathematica.MathematicaFunction`(*parent, name*)

Bases: `sage.interfaces.expect.ExpectFunction`

class `sage.interfaces.mathematica.MathematicaFunctionElement`(*obj, name*)

Bases: `sage.interfaces.expect.FunctionElement`

`sage.interfaces.mathematica.clean_output`(*s*)

`sage.interfaces.mathematica.mathematica_console`(*readline=True*)

`sage.interfaces.mathematica.parse_moutput_from_json`(*page_data, verbose=False*)

Return the list of outputs found in the json (with key 'moutput')

INPUT:

- `page_data` – json obtained from Wolfram Alpha
- `verbose` – bool (default: False)

OUTPUT:

list of unicode strings

EXAMPLES:

```
sage: from sage.interfaces.mathematica import request_wolfram_alpha
sage: from sage.interfaces.mathematica import parse_moutput_from_json
sage: page_data = request_wolfram_alpha('integrate Sin[x]') # optional internet
sage: parse_moutput_from_json(page_data) # optional internet
['-Cos[x]']
```

```
sage: page_data = request_wolfram_alpha('Sin[x]') # optional internet
sage: L = parse_moutput_from_json(page_data) # optional internet
sage: sorted(L) # optional internet
['-Cos[x]', '{x == 0}', '{x == Pi C[1], Element[C[1], Integers]}']
```

`sage.interfaces.mathematica.reduce_load`(*X*)

`sage.interfaces.mathematica.request_wolfram_alpha(input, verbose=False)`
Request Wolfram Alpha website.

INPUT:

- `input` – string
- `verbose` – bool (default: False)

OUTPUT:

json

EXAMPLES:

```
sage: from sage.interfaces.mathematica import request_wolfram_alpha
sage: page_data = request_wolfram_alpha('integrate Sin[x]') # optional internet
sage: [str(a) for a in sorted(page_data.keys())] # optional internet
['queryresult']
sage: [str(a) for a in sorted(page_data['queryresult'].keys())] # optional internet
['datatypes',
 'encryptedEvaluatedExpression',
 'encryptedParsedExpression',
 'error',
 'host',
 'id',
 'inputstring',
 'numpods',
 'parsetimedout',
 'parsetiming',
 'pods',
 'recalculate',
 'related',
 'server',
 'sponsorCategories',
 'success',
 'timedout',
 'timedoutpods',
 'timing',
 'version']
```

`sage.interfaces.mathematica.symbolic_expression_from_mathematica_string(mexpr)`
Translate a mathematica string into a symbolic expression

INPUT:

- `mexpr` – string

OUTPUT:

symbolic expression

EXAMPLES:

```
sage: from sage.interfaces.mathematica import symbolic_expression_from_mathematica_
↪string
sage: symbolic_expression_from_mathematica_string('-Cos[x]')
-cos(x)
```


INTERFACE TO MATHICS

Mathics is an open source interpreter for the Wolfram Language. From the introduction of its reference manual:

Note: Mathics — to be pronounced like “Mathematics” without the “emat” — is a general-purpose computer algebra system (CAS). It is meant to be a free, light-weight alternative to Mathematica®. It is free both as in “free beer” and as in “freedom”. There are various online mirrors running Mathics but it is also possible to run Mathics locally. A list of mirrors can be found at the Mathics homepage, <http://mathics.github.io>.

The programming language of Mathics is meant to resemble Wolfram’s famous Mathematica® as much as possible. However, Mathics is in no way affiliated or supported by Wolfram. Mathics will probably never have the power to compete with Mathematica® in industrial applications; yet, it might be an interesting alternative for educational purposes.

The Mathics interface will only work if the optional Sage package Mathics is installed. The interface lets you send certain Sage objects to Mathics, run Mathics functions, import certain Mathics expressions to Sage, or any combination of the above.

To send a Sage object `sobj` to Mathics, call `mathics(sobj)`. This exports the Sage object to Mathics and returns a new Sage object wrapping the Mathics expression/variable, so that you can use the Mathics variable from within Sage. You can then call Mathics functions on the new object; for example:

```
sage: from sage.interfaces.mathics import mathics
sage: mobj = mathics(x^2-1); mobj          # optional - mathics
-1 + x ^ 2
sage: mobj.Factor()                      # optional - mathics
(-1 + x) (1 + x)
```

In the above example the factorization is done using Mathics’s `Factor[]` function.

To see Mathics’s output you can simply print the Mathics wrapper object. However if you want to import Mathics’s output back to Sage, call the Mathics wrapper object’s `sage()` method. This method returns a native Sage object:

```
sage: mobj = mathics(x^2-1)                # optional - mathics
sage: mobj2 = mobj.Factor(); mobj2        # optional - mathics
(-1 + x) (1 + x)
sage: mobj2.parent()                      # optional - mathics
Mathics
sage: sobj = mobj2.sage(); sobj           # optional - mathics
(x + 1)*(x - 1)
sage: sobj.parent()                       # optional - mathics
Symbolic Ring
```

If you want to run a Mathics function and don’t already have the input in the form of a Sage object, then it might be simpler to input a string to `mathics(expr)`. This string will be evaluated as if you had typed it into Mathics:

```
sage: mathics('Factor[x^2-1]')          # optional - mathics
(-1 + x) (1 + x)
sage: mathics('Range[3]')              # optional - mathics
{1, 2, 3}
```

If you want work with the internal Mathics expression, then you can call `mathics.eval(expr)`, which returns an instance of `mathics.core.expression.Expression`. If you want the result to be a string formatted like Mathics's `InputForm`, call `repr(mobj)` on the wrapper object `mobj`. If you want a string formatted in Sage style, call `mobj._sage_repr()`:

```
sage: mathics.eval('x^2 - 1')           # optional - mathics
'-1 + x ^ 2'
sage: repr(mathics('Range[3]'))        # optional - mathics
'{1, 2, 3}'
sage: mathics('Range[3]')._sage_repr() # optional - mathics
'[1, 2, 3]'
```

Finally, if you just want to use a Mathics command line from within Sage, the function `mathics_console()` dumps you into an interactive command-line Mathics session.

26.1 Tutorial

We follow some of the tutorial from <http://library.wolfram.com/conferences/devconf99/withoff/Basic1.html/>.

26.1.1 Syntax

Now make 1 and add it to itself. The result is a Mathics object.

```
sage: m = mathics
sage: a = m(1) + m(1); a                # optional - mathics
2
sage: a.parent()                        # optional - mathics
Mathics
sage: m('1+1')                          # optional - mathics
2
sage: m(3)**m(50)                       # optional - mathics
717897987691852588770249
```

The following is equivalent to `Plus[2, 3]` in Mathics:

```
sage: m = mathics
sage: m(2).Plus(m(3))                   # optional - mathics
5
```

We can also compute $7(2 + 3)$.

```
sage: m(7).Times(m(2).Plus(m(3)))      # optional - mathics
35
sage: m('7(2+3)')                     # optional - mathics
35
```

26.1.2 Some typical input

We solve an equation and a system of two equations:

```
sage: eqn = mathics('3x + 5 == 14')      # optional - mathics
sage: eqn                               # optional - mathics
5 + 3 x == 14
sage: eqn.Solve('x')                   # optional - mathics
{{x -> 3}}
```

```
sage: sys = mathics('{x^2 - 3y == 3, 2x - y == 1}') # optional - mathics
sage: print(sys)                       # optional - mathics
{x ^ 2 - 3 y == 3, 2 x - y == 1}
sage: sys.Solve('{x, y}')              # optional - mathics
{{x -> 0, y -> -1}, {x -> 6, y -> 11}}
```

26.1.3 Assignments and definitions

If you assign the mathics 5 to a variable *c* in Sage, this does not affect the *c* in Mathics.

```
sage: c = m(5)                          # optional - mathics
sage: print(m('b + c x'))               # optional - mathics
      b + c x
sage: print(m('b') + c*m('x'))         # optional - mathics
      b + 5 x
```

The Sage interfaces changes Sage lists into Mathics lists:

```
sage: m = mathics
sage: eq1 = m('x^2 - 3y == 3')          # optional - mathics
sage: eq2 = m('2x - y == 1')           # optional - mathics
sage: v = m([eq1, eq2]); v              # optional - mathics
{x ^ 2 - 3 y == 3, 2 x - y == 1}
sage: v.Solve(['x', 'y'])               # optional - mathics
{{x -> 0, y -> -1}, {x -> 6, y -> 11}}
```

26.1.4 Function definitions

Define mathics functions by simply sending the definition to the interpreter.

```
sage: m = mathics
sage: _ = mathics('f[p_] = p^2');      # optional - mathics
sage: m('f[9]')                        # optional - mathics
81
```

26.1.5 Numerical Calculations

We find the x such that $e^x - 3x = 0$.

```
sage: eqn = mathics('Exp[x] - 3x == 0') # optional - mathics
sage: eqn.FindRoot(['x', 2])           # optional - mathics
{x -> 1.51213}
```

Note that this agrees with what the PARI interpreter gp produces:

```
sage: gp('solve(x=1,2,exp(x)-3*x)')
1.512134551657842473896739678      # 32-bit
1.5121345516578424738967396780720387046 # 64-bit
```

Next we find the minimum of a polynomial using the two different ways of accessing Mathics:

```
sage: mathics('FindMinimum[x^3 - 6x^2 + 11x - 5, {x,3}]') # not tested (since not_
↳supported, so far)
{0.6150998205402516, {x -> 2.5773502699629733}}
sage: f = mathics('x^3 - 6x^2 + 11x - 5')                # optional - mathics
sage: f.FindMinimum(['x', 3])                             # not tested (since not_
↳supported, so far)
{0.6150998205402516, {x -> 2.5773502699629733}}
```

26.1.6 Polynomial and Integer Factorization

We factor a polynomial of degree 200 over the integers.

```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = (x**100+17*x+5)*(x**100-5*x+20)
sage: f
x^200 + 12*x^101 + 25*x^100 - 85*x^2 + 315*x + 100
sage: g = mathics(str(f)) # optional - mathics
sage: print(g)           # optional - mathics
100 + 315 x - 85 x ^ 2 + 25 x ^ 100 + 12 x ^ 101 + x ^ 200
sage: g                  # optional - mathics
100 + 315 x - 85 x ^ 2 + 25 x ^ 100 + 12 x ^ 101 + x ^ 200
sage: print(g.Factor()) # optional - mathics
(5 + 17 x + x ^ 100) (20 - 5 x + x ^ 100)
```

We can also factor a multivariate polynomial:

```
sage: f = mathics('x^6 + (-y - 2)*x^5 + (y^3 + 2*y)*x^4 - y^4*x^3') # optional - mathics
sage: print(f.Factor()) # optional - mathics
x ^ 3 (x - y) (-2 x + x ^ 2 + y ^ 3)
```

We factor an integer:

```
sage: n = mathics(2434500) # optional - mathics
sage: n.FactorInteger()  # optional - mathics
{{2, 2}, {3, 2}, {5, 3}, {541, 1}}
sage: n = mathics(2434500) # optional - mathics
sage: F = n.FactorInteger(); F # optional - mathics
```

(continues on next page)

(continued from previous page)

```

{{2, 2}, {3, 2}, {5, 3}, {541, 1}}
sage: F[1] # optional - mathics
{2, 2}
sage: F[4] # optional - mathics
{541, 1}

```

26.2 Long Input

The Mathics interface reads in even very long input (using files) in a robust manner.

```

sage: t = "%s"%10^10000 # ten thousand character string.
sage: a = mathics(t) # optional - mathics
sage: a = mathics.eval(t) # optional - mathics

```

26.3 Loading and saving

Mathics has an excellent `InputForm` function, which makes saving and loading Mathics objects possible. The first examples test saving and loading to strings.

```

sage: x = mathics(pi/2) # optional - mathics
sage: print(x) # optional - mathics
Pi / 2
sage: loads(dumps(x)) == x # optional - mathics
True
sage: n = x.N(50) # optional - mathics
sage: print(n) # optional - mathics
1.5707963267948966192313216916397514420985846996876
sage: loads(dumps(n)) == n # optional - mathics
True

```

26.4 Complicated translations

The `obj.sage()` method tries to convert a Mathics object to a Sage object. In many cases, it will just work. In particular, it should be able to convert expressions entirely consisting of:

- numbers, i.e. integers, floats, complex numbers;
- functions and named constants also present in Sage, where:
 - Sage knows how to translate the function or constant's name from Mathics's, or
 - the Sage name for the function or constant is trivially related to Mathics's;
- symbolic variables whose names don't pathologically overlap with objects already defined in Sage.

This method will not work when Mathics's output includes:

- strings;
- functions unknown to Sage;

- Mathics functions with different parameters/parameter order to the Sage equivalent.

If you want to convert more complicated Mathics expressions, you can instead call `mobj._sage_()` and supply a translation dictionary:

```
sage: x = var('x')
sage: m = mathics('NewFn[x]') # optional - mathics
sage: m._sage_(locals={'NewFn': sin, 'x':x}) # optional - mathics
sin(x)
```

For more details, see the documentation for `._sage_()`.

OTHER Examples:

```
sage: def math_bessel_K(nu,x):
.....:     return mathics(nu).BesselK(x).N(20)
sage: math_bessel_K(2,I) # optional - mathics
-2.5928861754911969782 + 0.18048997206696202663 I
```

```
sage: slist = [[1, 2], 3., 4 + I]
sage: mlist = mathics(slist); mlist # optional - mathics
{{1, 2}, 3., 4 + I}
sage: slist2 = list(mlist); slist2 # optional - mathics
[[1, 2], 3., 4 + I]
sage: slist2[0] # optional - mathics
{1, 2}
sage: slist2[0].parent() # optional - mathics
Mathics
sage: slist3 = mlist.sage(); slist3 # optional - mathics
[[1, 2], 3.0000000000000000, 4.0000000000000000 + 1.0000000000000000*I]
```

```
sage: mathics('10.^80') # optional - mathics
1.*^80
sage: mathics('10.^80').sage() # optional - mathics
1.0000000000000000e80
```

AUTHORS:

- Sebastian Oehms (2021): first version from a copy of the Mathematica interface (see [trac ticket #31778](#)).

Thanks to Rocky Bernstein and Juan Mauricio Matera for their support. For further acknowledgments see [this list](#).

class `sage.interfaces.mathics.Mathics` (*maxread=None, logfile=None, init_list_length=1024, seed=None*)

Bases: `sage.interfaces.interface.Interface`

Interface to the Mathics interpreter.

Implemented according to the Mathematica interface but avoiding Pexpect functionality.

EXAMPLES:

```
sage: t = mathics('Tan[I + 0.5]') # optional - mathics
sage: t.parent() # optional - mathics
Mathics
sage: ts = t.sage() # optional - mathics
sage: ts.parent() # optional - mathics
Complex Field with 53 bits of precision
```

(continues on next page)

(continued from previous page)

```

sage: t == mathics(ts)           # optional - mathics
True
sage: mtan = mathics.Tan         # optional - mathics
sage: mt = mtan(I+1/2)          # optional - mathics
sage: mt == t                   # optional - mathics
True
sage: u = mathics(I+1/2)        # optional - mathics
sage: u.Tan() == mt             # optional - mathics
True

```

More examples can be found in the module header.

chdir(*dir*)

Change Mathics's current working directory.

EXAMPLES:

```

sage: mathics.chdir('/')         # optional - mathics
sage: mathics('Directory[]')   # optional - mathics
/

```

console()

Spawn a new Mathics command-line session.

EXAMPLES:

```

sage: mathics.console() # not tested

Mathics 2.1.1.dev0
on CPython 3.9.2 (default, Mar 19 2021, 22:23:28)
using SymPy 1.7, mpmath 1.2.1, numpy 1.19.5, cython 0.29.21

Copyright (C) 2011-2021 The Mathics Team.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions.
See the documentation for the full license.

Quit by evaluating Quit[] or by pressing CONTROL-D.

In[1]:= Sin[0.5]
Out[1]= 0.479426

Goodbye!

sage:

```

eval(*code*, **args*, ***kwds*)

Evaluates a command inside the Mathics interpreter and returns the output in printable form.

EXAMPLES:

```

sage: mathics.eval('1+1') # optional - mathics
'2'

```

get(*var*)

Get the value of the variable *var*.

EXAMPLES:

```
sage: mathics.set('u', '2*x +E')           # optional - mathics
sage: mathics.get('u')                     # optional - mathics
'E + 2 x'
```

help(*cmd*, *long=False*)

Return the Mathics documentation of the given command.

EXAMPLES:

```
sage: mathics.help('Sin')                  # optional - mathics
"\n 'Sin[z]'\n  returns the sine of z.\n"

sage: print(_)                            # optional - mathics

'Sin[z]'
  returns the sine of z.

sage: print(mathics.help('Sin', long=True)) # optional - mathics

'Sin[z]'
  returns the sine of z.

Attributes[Sin] = {Listable, NumericFunction, Protected}

sage: print(mathics.Factorial.__doc__)    # optional - mathics

'Factorial[n]'
'n!'
  computes the factorial of n.

sage: u = mathics('Pi')                  # optional - mathics
sage: print(u.Cos.__doc__)                # optional - mathics

'Cos[z]'
  returns the cosine of z.
```

set(*var*, *value*)

Set the variable *var* to the given value.

EXAMPLES:

```
sage: mathics.set('u', '2*x +E')           # optional - mathics
sage: bool(mathics('u').sage() == 2*x+E) # optional - mathics
True
```

```
class sage.interfaces.mathics.MathicsElement(parent, value, is_name=False, name=None)
  Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.interface.InterfaceElement
```

Element class of the Mathics interface.

Its instances are usually constructed via the instance call of its parent. It wraps the Mathics library for this object. In a session Mathics methods can be obtained using tab completion.

EXAMPLES:

```
sage: me=mathics(e); me           # optional - mathics
E
sage: type(me)                   # optional - mathics
<class 'sage.interfaces.mathics.MathicsElement'>
sage: P = me.parent(); P        # optional - mathics
Mathics
sage: type(P)                   # optional - mathics
<class 'sage.interfaces.mathics.Mathics'>
```

Access to the Mathics expression objects:

```
sage: res = me._mathics_result   # optional - mathics
sage: type(res)                 # optional - mathics
<class 'mathics.core.evaluation.Result'>
sage: expr = res.last_eval; expr # optional - mathics
<Symbol: System`E>
sage: type(expr)               # optional - mathics
<class 'mathics.core.expression.Symbol'>
```

Applying Mathics methods:

```
sage: me.to_sympy()             # optional - mathics
E
sage: me.get_name()             # optional - mathics
'System`E'
sage: me.is_inexact()          # optional - mathics
False
sage: me.is_symbol()           # optional - mathics
True
```

Conversion to Sage:

```
sage: bool(me.sage() == e)      # optional - mathics
True
```

`n(*args, **kwargs)`

Numerical approximation by converting to Sage object first

Convert the object into a Sage object and return its numerical approximation. See documentation of the function `sage.misc.functional.n()` for details.

EXAMPLES:

```
sage: mathics('Pi').n(10)      # optional -- mathics
3.1
sage: mathics('Pi').n()        # optional -- mathics
3.14159265358979
sage: mathics('Pi').n(digits=10) # optional -- mathics
3.141592654
```

save_image(filename, ImageSize=600)

Save a mathics graphics

INPUT:

- filename – string. The filename to save as. The extension determines the image file format.
- ImageSize – integer. The size of the resulting image.

EXAMPLES:

```
sage: P = mathics('Plot[Sin[x],{x,-2Pi,4Pi}']') # optional - mathics
sage: filename = tmp_filename() # optional - mathics
sage: P.save_image(filename, ImageSize=800) # optional - mathics
```

show(ImageSize=600)

Show a mathics expression immediately.

This method attempts to display the graphics immediately, without waiting for the currently running code (if any) to return to the command line. Be careful, calling it from within a loop will potentially launch a large number of external viewer programs.

INPUT:

- ImageSize – integer. The size of the resulting image.

OUTPUT:

This method does not return anything. Use save() if you want to save the figure as an image.

EXAMPLES:

```
sage: Q = mathics('Sin[x Cos[y]]/Sqrt[1-x^2]') # optional - mathics
sage: show(Q) # optional - mathics
Sin[x Cos[y]] / Sqrt[1 - x ^ 2]

sage: P = mathics('Plot[Sin[x],{x,-2Pi,4Pi}']') # optional - mathics
sage: show(P) # optional - mathics
sage: P.show(ImageSize=800) # optional - mathics
```

sage.interfaces.mathics.**mathics_console**()

Spawn a new Mathics command-line session.

EXAMPLES:

```
sage: mathics_console() # not tested

Mathics 2.1.1.dev0
on CPython 3.9.2 (default, Mar 19 2021, 22:23:28)
using SymPy 1.7, mpmath 1.2.1, numpy 1.19.5, cython 0.29.21

Copyright (C) 2011-2021 The Mathics Team.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions.
See the documentation for the full license.

Quit by evaluating Quit[] or by pressing CONTROL-D.
```

(continues on next page)

(continued from previous page)

```
In[1]:= Sin[0.5]
Out[1]= 0.479426

Goodbye!
```

`sage.interfaces.mathics.reduce_load(X)`

Used in unpickling a Mathics element.

This function is just the `__call__` method of the interface instance.

EXAMPLES:

```
sage: sage.interfaces.mathics.reduce_load('Denominator[a / b]') # optional --
↳mathics
b
```


INTERFACE TO MATLAB

According to their website, MATLAB is “a high-level language and interactive environment that enables you to perform computationally intensive tasks faster than with traditional programming languages such as C, C++, and Fortran.”

The commands in this section only work if you have the “matlab” interpreter installed and available in your PATH. It’s not necessary to install any special Sage packages.

EXAMPLES:

```
sage: matlab.eval('2+2')           # optional - matlab
'\nans =\n\n      4\n'
```

```
sage: a = matlab(10)              # optional - matlab
sage: a**10                       # optional - matlab
      1.0000e+10
```

AUTHORS:

- William Stein (2006-10-11)

27.1 Tutorial

EXAMPLES:

```
sage: matlab('4+10')             # optional - matlab
14
sage: matlab('date')             # optional - matlab; random output
18-Oct-2006
sage: matlab('5*10 + 6')         # optional - matlab
56
sage: matlab('(6+6)/3')          # optional - matlab
4
sage: matlab('9')^2              # optional - matlab
81
sage: a = matlab(10); b = matlab(20); c = matlab(30)   # optional - matlab
sage: avg = (a+b+c)/3 ; avg      # optional - matlab
20
sage: parent(avg)                # optional - matlab
Matlab
```

```

sage: my_scalar = matlab('3.1415')      # optional - matlab
sage: my_scalar                          # optional - matlab
3.1415
sage: my_vector1 = matlab('[1,5,7]')    # optional - matlab
sage: my_vector1                        # optional - matlab
1    5    7
sage: my_vector2 = matlab('[1;5;7]')    # optional - matlab
sage: my_vector2                        # optional - matlab
1
5
7
sage: my_vector1 * my_vector2           # optional - matlab
75

```

```

sage: row_vector1 = matlab('[1 2 3]')    # optional - matlab
sage: row_vector2 = matlab('[3 2 1]')    # optional - matlab
sage: matrix_from_row_vec = matlab('%s; %s'%(row_vector1.name(), row_vector2.name())) ↵
↪ # optional - matlab
sage: matrix_from_row_vec                # optional - matlab
1    2    3
3    2    1

```

```

sage: column_vector1 = matlab('[1;3]')    # optional - matlab
sage: column_vector2 = matlab('[2;8]')    # optional - matlab
sage: matrix_from_col_vec = matlab('%s %s'%(column_vector1.name(), column_vector2.
↪name()))                                # optional - matlab
sage: matrix_from_col_vec                # optional - matlab
1    2
3    8

```

```

sage: my_matrix = matlab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]') # optional ↵
↪matlab
sage: my_matrix                          # optional - matlab
8    12   19
7    3    2
12   4   23
8    1    1

```

```

sage: combined_matrix = matlab('%s, %s'%(my_matrix.name(), my_matrix.name())) ↵
↪
# optional - matlab
sage: combined_matrix                    # optional - matlab
8    12   19    8    12   19
7    3    2    7    3    2
12   4   23   12   4   23
8    1    1    8    1    1

```

```

sage: tm = matlab('0.5:2:10')            # optional - matlab
sage: tm                                  # optional - matlab
0.5000    2.5000    4.5000    6.5000    8.5000

```

```
sage: my_vector1 = matlab('[1,5,7]')           # optional - matlab
sage: my_vector1(1)                           # optional - matlab
1
sage: my_vector1(2)                           # optional - matlab
5
sage: my_vector1(3)                           # optional - matlab
7
```

Matrix indexing works as follows:

```
sage: my_matrix = matlab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]') # optional - ↵
↵matlab
sage: my_matrix(3,2)                          # optional - matlab
4
```

Setting using parenthesis cannot work (because of how the Python language works). Use square brackets or the set function:

```
sage: my_matrix = matlab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]') # optional - ↵
↵matlab
sage: my_matrix.set(2,3, 1999)                # optional - matlab
sage: my_matrix                               # optional - matlab
      8      12      19
      7       3     1999
     12       4      23
      8       1       1
```

class `sage.interfaces.matlab.Matlab`(*maxread=None, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None*)

Bases: `sage.interfaces.expect.Expect`

Interface to the Matlab interpreter.

EXAMPLES:

```
sage: a = matlab('[ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]') # optional - matlab
sage: b = matlab('[ 1; 3; 13]')                   # optional - matlab
sage: c = a * b                                   # optional - matlab
sage: print(c)                                    # optional - matlab
      30
     122
     505
```

chdir(*directory*)

Change MATLAB's current working directory.

EXAMPLES:

```
sage: matlab.chdir('/')                          # optional - matlab
sage: matlab.pwd()                              # optional - matlab
/
```

console()

get(*var*)

Get the value of the variable *var*.

EXAMPLES:

```
sage: s = matlab.eval('a = 2') # optional - matlab
sage: matlab.get('a')          # optional - matlab
' 2'
```

sage2matlab_matrix_string(A)

Return an matlab matrix from a Sage matrix.

INPUT: A Sage matrix with entries in the rationals or reals.

OUTPUT: A string that evaluates to an Matlab matrix.

EXAMPLES:

```
sage: M33 = MatrixSpace(QQ,3,3)
sage: A = M33([1,2,3,4,5,6,7,8,0])
sage: matlab.sage2matlab_matrix_string(A) # optional - matlab
'[1, 2, 3; 4, 5, 6; 7, 8, 0]'
```

AUTHOR:

- David Joyner and William Stein

set(var, value)

Set the variable var to the given value.

strip_answer(s)

Returns the string s with Matlab’s answer prompt removed.

EXAMPLES:

```
sage: s = '\nans =\n\n 2\n'
sage: matlab.strip_answer(s)
' 2'
```

version()

whos()

class sage.interfaces.matlab.MatlabElement(parent, value, is_name=False, name=None)

Bases: *sage.interfaces.expect.ExpectElement*

set(i, j, x)

sage.interfaces.matlab.matlab_console()

This requires that the optional matlab program be installed and in your PATH, but no optional Sage packages need be installed.

EXAMPLES:

```
sage: matlab_console() # optional - matlab; not tested
      < M A T L A B >
      Copyright 1984-2006 The MathWorks, Inc.
...
>> 2+3
```

ans =

5

quit

Typing `quit` exits the matlab console and returns you to Sage. `matlab`, like Sage, remembers its history from one session to another.

`sage.interfaces.matlab.matlab_version()`

Return the version of Matlab installed.

EXAMPLES:

```
sage: matlab_version()    # random; optional - matlab
'7.2.0.283 (R2006a)'
```

`sage.interfaces.matlab.reduce_load_Matlab()`

PEXPECT INTERFACE TO MAXIMA

Maxima is a free GPL'd general purpose computer algebra system whose development started in 1968 at MIT. It contains symbolic manipulation algorithms, as well as implementations of special functions, including elliptic functions and generalized hypergeometric functions. Moreover, Maxima has implementations of many functions relating to the invariant theory of the symmetric group S_n . (However, the commands for group invariants, and the corresponding Maxima documentation, are in French.) For many links to Maxima documentation see <http://maxima.sourceforge.net/documentation.html>.

AUTHORS:

- William Stein (2005-12): Initial version
- David Joyner: Improved documentation
- William Stein (2006-01-08): Fixed bug in parsing
- William Stein (2006-02-22): comparisons (following suggestion of David Joyner)
- William Stein (2006-02-24): *greatly* improved robustness by adding sequence numbers to IO bracketing in `_eval_line`
- Robert Bradshaw, Nils Bruin, Jean-Pierre Flori (2010,2011): Binary library interface

This is the interface used by the maxima object:

```
sage: type(maxima)
<class 'sage.interfaces.maxima.Maxima'>
```

If the string “error” (case insensitive) occurs in the output of anything from Maxima, a `RuntimeError` exception is raised.

EXAMPLES: We evaluate a very simple expression in Maxima.

```
sage: maxima('3 * 5')
15
```

We factor $x^5 - y^5$ in Maxima in several different ways. The first way yields a Maxima object.

```
sage: F = maxima.factor('x^5 - y^5')
sage: F
-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4)
sage: type(F)
<class 'sage.interfaces.maxima.MaximaElement'>
```

Note that Maxima objects can also be displayed using “ASCII art”; to see a normal linear representation of any Maxima object `x`. Just use the print command: use `str(x)`.

```
sage: print(F)
      4      3      2      2      3      4
-(y - x)(y + x y + x y + x y + x )
```

You can always use `repr(x)` to obtain the linear representation of an object. This can be useful for moving maxima data to other systems.

```
sage: repr(F)
'-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4) '
sage: F.str()
'-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4) '
```

The `maxima.eval` command evaluates an expression in maxima and returns the result as a *string* not a maxima object.

```
sage: print(maxima.eval('factor(x^5 - y^5)'))
-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4)
```

We can create the polynomial f as a Maxima polynomial, then call the `factor` method on it. Notice that the notation `f.factor()` is consistent with how the rest of Sage works.

```
sage: f = maxima('x^5 - y^5')
sage: f^2
(x^5-y^5)^2
sage: f.factor()
-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4)
```

Control-C interruption works well with the maxima interface, because of the excellent implementation of maxima. For example, try the following sum but with a much bigger range, and hit control-C.

```
sage: maxima('sum(1/x^2, x, 1, 10)')
1968329/1270080
```

28.1 Tutorial

We follow the tutorial at <http://maxima.sourceforge.net/docs/intromax/intromax.html>.

```
sage: maxima('1/100 + 1/101')
201/10100
```

```
sage: a = maxima('(1 + sqrt(2))^5'); a
(sqrt(2)+1)^5
sage: a.expand()
29*sqrt(2)+41
```

```
sage: a = maxima('(1 + sqrt(2))^5')
sage: float(a)
82.0121933088197...
sage: a.numer()
82.0121933088197...
```



```

sage: f = maxima('x^3 * %e^(k*x) * sin(w*x)'); f
x^3*e^(k*x)*sin(w*x)
sage: f.diff('x')
k*x^3*e^(k*x)*sin(w*x)+3*x^2*e^(k*x)*sin(w*x)+w*x^3*e^(k*x) *cos(w*x)
sage: f.integrate('x')
(((k*w^6+3*k^3*w^4+3*k^5*w^2+k^7)*x^3 + (3*w^6+3*k^2*w^4-3*k^4*w^2-3*k^6)*x^2+((-18*k*w^
↪ 4)-12*k^3*w^2+6*k^5)*x-6*w^4 +36*k^2*w^2-6*k^4) *%e^(k*x)*sin(w*x) + (((-w^7)-3*k^2*w^5-
↪ 3*k^4*w^3-k^6*w)*x^3 + (6*k*w^5+12*k^3*w^3+6*k^5*w)*x^2+(6*w^5-12*k^2*w^3-18*k^4*w)*x-
↪ 24*k*w^3 +24*k^3*w) *%e^(k*x)*cos(w*x)) / (w^8+4*k^2*w^6+6*k^4*w^4+4*k^6*w^2+k^8)
    
```

```

sage: f = maxima('1/x^2')
sage: f.integrate('x', 1, 'inf')
1
sage: g = maxima('f/sinh(k*x)^4')
sage: g.taylor('x', 0, 3)
f/(k^4*x^4)-(2*f)/((3*k^2)*x^2)+(11*f)/45-((62*k^2*f)*x^2)/945
    
```

```

sage: maxima.taylor('asin(x)', 'x', 0, 10)
x+x^3/6+(3*x^5)/40+(5*x^7)/112+(35*x^9)/1152
    
```

28.2 Examples involving matrices

We illustrate computing with the matrix whose i, j entry is i/j , for $i, j = 1, \dots, 4$.

```

sage: f = maxima.eval('f[i,j] := i/j')
sage: A = maxima('genmatrix(f,4,4)'); A
matrix([[1, 1/2, 1/3, 1/4], [2, 1, 2/3, 1/2], [3, 3/2, 1, 3/4], [4, 2, 4/3, 1]])
sage: A.determinant()
0
sage: A.echelon()
matrix([[1, 1/2, 1/3, 1/4], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]])
sage: A.eigenvalues()
[[0, 4], [3, 1]]
sage: A.eigenvectors()
[[[0, 4], [3, 1]], [[1, 0, 0, -4], [0, 1, 0, -2], [0, 0, 1, -4/3]], [[1, 2, 3, 4]]]
    
```

We can also compute the echelon form in Sage:

```

sage: B = matrix(QQ, A)
sage: B.echelon_form()
[ 1 1/2 1/3 1/4]
[ 0 0 0 0]
[ 0 0 0 0]
[ 0 0 0 0]
sage: B.charpoly('x').factor()
(x - 4) * x^3
    
```

28.3 Laplace Transforms

We illustrate Laplace transforms:

```
sage: _ = maxima.eval("f(t) := t*sin(t)")
sage: maxima("laplace(f(t),t,s)")
(2*s)/(s^2+1)^2
```

```
sage: maxima("laplace(delta(t-3),t,s)") #Dirac delta function
%e^-(3*s)
```

```
sage: _ = maxima.eval("f(t) := exp(t)*sin(t)")
sage: maxima("laplace(f(t),t,s)")
1/(s^2-2*s+2)
```

```
sage: _ = maxima.eval("f(t) := t^5*exp(t)*sin(t)")
sage: maxima("laplace(f(t),t,s)")
(360*(2*s-2))/(s^2-2*s+2)^4-(480*(2*s-2)^3)/(s^2-2*s+2)^5 +(120*(2*s-2)^5)/(s^2-2*s+2)^6
sage: print(maxima("laplace(f(t),t,s)"))
          3              5
    360 (2 s - 2)    480 (2 s - 2)    120 (2 s - 2)
    ----- - ----- + -----
          2            4            2            5            2            6
    (s  - 2 s + 2)  (s  - 2 s + 2)  (s  - 2 s + 2)
```

```
sage: maxima("laplace(diff(x(t),t),t,s)")
s*'laplace(x(t),t,s)-x(0)
```

```
sage: maxima("laplace(diff(x(t),t,2),t,s)")
(-%at('diff(x(t),t,1),t = 0))+s^2*'laplace(x(t),t,s)-x(0)*s
```

It is difficult to read some of these without the 2d representation:

```
sage: print(maxima("laplace(diff(x(t),t,2),t,s)"))
          !
          d      !      2
    (- -- (x(t))!      ) + s  laplace(x(t), t, s) - x(0) s
          dt      !
          !t = 0
```

Even better, use `view(maxima("laplace(diff(x(t),t,2),t,s)"))` to see a typeset version.

28.4 Continued Fractions

A continued fraction $a + 1/(b + 1/(c + \dots))$ is represented in maxima by the list $[a, b, c, \dots]$.

```
sage: maxima("cf((1 + sqrt(5))/2)")
[1, 1, 1, 1, 2]
sage: maxima("cf ((1 + sqrt(341))/2)")
[9, 1, 2, 1, 2, 1, 17, 1, 2, 1, 2, 1, 17, 1, 2, 1, 2, 1, 17, 2]
```

28.5 Special examples

In this section we illustrate calculations that would be awkward to do (as far as I know) in non-symbolic computer algebra systems like MAGMA or GAP.

We compute the gcd of $2x^{n+4} - x^{n+2}$ and $4x^{n+1} + 3x^n$ for arbitrary n .

```
sage: f = maxima('2*x^(n+4) - x^(n+2)')
sage: g = maxima('4*x^(n+1) + 3*x^n')
sage: f.gcd(g)
x^n
```

You can plot 3d graphs (via gnuplot):

```
sage: maxima('plot3d(x^2-y^2, [x,-2,2], [y,-2,2], [grid,12,12])') # not tested
[displays a 3 dimensional graph]
```

You can formally evaluate sums (note the nusum command):

```
sage: S = maxima('nusum(exp(1+2*i/n), i, 1, n)')
sage: print(S)
```

$$\frac{e^{2/n+3}}{e^{1/n}-1} \frac{e^{2/n+1}}{e^{1/n}+1}$$

We formally compute the limit as $n \rightarrow \infty$ of $2S/n$ as follows:

```
sage: T = S*maxima('2/n')
sage: T.tlimit('n', 'inf')
%e^3-%e
```

28.6 Miscellaneous

Obtaining digits of π :

```
sage: maxima.eval('fpprec : 100')
'100'
sage: maxima(pi).bfloat()
3.
↪ 141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117068b0
```

Defining functions in maxima:

```
sage: maxima.eval('fun[a] := a^2')
'fun[a]:=a^2'
sage: maxima('fun[10]')
100
```

28.7 Interactivity

Unfortunately maxima doesn't seem to have a non-interactive mode, which is needed for the Sage interface. If any Sage call leads to maxima interactively answering questions, then the questions can't be answered and the maxima session may hang. See the discussion at <http://www.ma.utexas.edu/pipermail/maxima/2005/011061.html> for some ideas about how to fix this problem. An example that illustrates this problem is `maxima.eval('integrate (exp(a*x), x, 0, inf)')`.

28.8 Latex Output

To TeX a maxima object do this:

```
sage: latex(maxima('sin(u) + sinh(v^2)'))
\sinh v^2+\sin u
```

Here's another example:

```
sage: g = maxima('exp(3*i*x)/(6*i) + exp(i*x)/(2*i) + c')
sage: latex(g)
-{{i\,e^{3\,i\,x}}\over{6}}-{{i\,e^{i\,x}}\over{2}}+c
```

28.9 Long Input

The MAXIMA interface reads in even very long input (using files) in a robust manner, as long as you are creating a new object.

Note: Using `maxima.eval` for long input is much less robust, and is not recommended.

```
sage: t = "%s"%10^10000 # ten thousand character string.
sage: a = maxima(t)
```

class `sage.interfaces.maxima.Maxima`(*script_subdirectory=None, logfile=None, server=None, init_code=None*)

Bases: `sage.interfaces.maxima_abstract.MaximaAbstract`, `sage.interfaces.expect.Expect`

Interface to the Maxima interpreter.

EXAMPLES:

```
sage: m = Maxima()
sage: m == maxima
False
```

clear(*var*)

Clear the variable named *var*.

EXAMPLES:

```
sage: maxima.set('xxxxx', '2')
sage: maxima.get('xxxxx')
'2'
sage: maxima.clear('xxxxx')
sage: maxima.get('xxxxx')
'xxxxx'
```

get(*var*)

Get the string value of the variable *var*.

EXAMPLES:

```
sage: maxima.set('xxxxx', '2')
sage: maxima.get('xxxxx')
'2'
```

lisp(*cmd*)

Send a lisp command to Maxima.

Note: The output of this command is very raw - not pretty.

EXAMPLES:

```
sage: maxima.lisp("(+ 2 17)") # random formatted output
:lisp (+ 2 17)
19
(
```

set(*var, value*)

Set the variable *var* to the given value.

INPUT:

- *var* - string
- *value* - string

EXAMPLES:

```
sage: maxima.set('xxxxx', '2')
sage: maxima.get('xxxxx')
'2'
```

set_seed(seed=None)

http://maxima.sourceforge.net/docs/manual/maxima_10.html `make_random_state` (n) returns a new random state object created from an integer seed value equal to n modulo 2^{32} . n may be negative.

EXAMPLES:

```
sage: m = Maxima()
sage: m.set_seed(1)
1
sage: [m.random(100) for i in range(5)]
[45, 39, 24, 68, 63]
```

class `sage.interfaces.maxima.MaximaElement`(parent, value, is_name=False, name=None)

Bases: `sage.interfaces.maxima_abstract.MaximaAbstractElement`, `sage.interfaces.expect.ExpectElement`

Element of Maxima through Pexpect interface.

EXAMPLES:

Elements of this class should not be created directly. The targeted parent should be used instead:

```
sage: maxima(3)
3
sage: maxima(cos(x)+e^234)
cos(_SAGE_VAR_x)+%e^234
```

display2d(onscreen=True)

Return the 2d string representation of this Maxima object.

EXAMPLES:

```
sage: F = maxima('x^5 - y^5').factor()
sage: F.display2d()
      4      3      2 2      3      4
    - (y - x) (y + x y + x y + x y + x )
```

class `sage.interfaces.maxima.MaximaElementFunction`(parent, name, defn, args, latex)

Bases: `sage.interfaces.maxima.MaximaElement`, `sage.interfaces.maxima_abstract.MaximaAbstractElementFunction`

Maxima user-defined functions.

EXAMPLES:

Elements of this class should not be created directly. The method `function` of the targeted parent should be used instead:

```
sage: maxima.function('x,y', 'h(x)*y')
h(x)*y
```

`sage.interfaces.maxima.is_MaximaElement`(x)

Returns True if x is of type `MaximaElement`.

EXAMPLES:

```
sage: from sage.interfaces.maxima import is_MaximaElement
sage: m = maxima(1)
sage: is_MaximaElement(m)
True
sage: is_MaximaElement(1)
False
```

`sage.interfaces.maxima.reduce_load_Maxima()`

Unpickle a Maxima Pexpect interface.

EXAMPLES:

```
sage: from sage.interfaces.maxima import reduce_load_Maxima
sage: reduce_load_Maxima()
Maxima
```

`sage.interfaces.maxima.reduce_load_Maxima_function(parent, defn, args, latex)`

Unpickle a Maxima function.

EXAMPLES:

```
sage: from sage.interfaces.maxima import reduce_load_Maxima_function
sage: f = maxima.function('x,y','sin(x+y)')
sage: _,args = f.__reduce__()
sage: g = reduce_load_Maxima_function(*args)
sage: g == f
True
```

ABSTRACT INTERFACE TO MAXIMA

Maxima is a free GPL'd general purpose computer algebra system whose development started in 1968 at MIT. It contains symbolic manipulation algorithms, as well as implementations of special functions, including elliptic functions and generalized hypergeometric functions. Moreover, Maxima has implementations of many functions relating to the invariant theory of the symmetric group S_n . (However, the commands for group invariants, and the corresponding Maxima documentation, are in French.) For many links to Maxima documentation see <http://maxima.sourceforge.net/docs.shtml/>.

AUTHORS:

- William Stein (2005-12): Initial version
- David Joyner: Improved documentation
- William Stein (2006-01-08): Fixed bug in parsing
- William Stein (2006-02-22): comparisons (following suggestion of David Joyner)
- William Stein (2006-02-24): *greatly* improved robustness by adding sequence numbers to IO bracketing in `_eval_line`
- Robert Bradshaw, Nils Bruin, Jean-Pierre Flori (2010,2011): Binary library interface

This is an abstract class implementing the functions shared between the Pexpect and library interfaces to Maxima.

class `sage.interfaces.maxima_abstract.MaximaAbstract`(*name='maxima_abstract'*)

Bases: `sage.interfaces.tab_completion.ExtraTabCompletion`, [*sage.interfaces.interface.Interface*](#)

Abstract interface to Maxima.

INPUT:

- `name` - string

OUTPUT: the interface

EXAMPLES:

This class should not be instantiated directly, but through its subclasses `Maxima` (Pexpect interface) or `MaximaLib` (library interface):

```
sage: m = Maxima()
sage: from sage.interfaces.maxima_abstract import MaximaAbstract
sage: isinstance(m, MaximaAbstract)
True
```

chdir(*dir*)

Change Maxima's current working directory.

INPUT:

- `dir` - string

OUTPUT: none

EXAMPLES:

```
sage: maxima.chdir('/')
```

completions(*s*, *verbose=True*)

Return all commands that complete the command starting with the string *s*. This is like typing *s*[tab] in the Maxima interpreter.

INPUT:

- *s* - string
- *verbose* - boolean (default: True)

OUTPUT: array of strings

EXAMPLES:

```
sage: sorted(maxima.completions('gc', verbose=False))
['gcd', 'gcdex', 'gcfactor', 'gctime']
```

console()

Start the interactive Maxima console. This is a completely separate maxima session from this interface. To interact with this session, you should instead use `maxima.interact()`.

INPUT: none

OUTPUT: none

EXAMPLES:

```
sage: maxima.console()           # not tested (since we can't)
Maxima 5.34.1 http://maxima.sourceforge.net
Using Lisp ECL 13.5.1
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
This is a development version of Maxima. The function bug_report()
provides bug reporting information.
(%i1)
```

```
sage: maxima.interact()        # this is not tested either
--> Switching to Maxima <--
maxima: 2+2
4
maxima:
--> Exiting back to Sage <--
```

cputime(*t=None*)

Returns the amount of CPU time that this Maxima session has used.

INPUT:

- *t* - float (default: None); If `var{t}` is not None, then it returns the difference between the current CPU time and `var{t}`.

OUTPUT: float

EXAMPLES:

```
sage: t = maxima.cputime()
sage: _ = maxima.de_solve('diff(y,x,2) + 3*x = y', ['x','y'], [1,1,1])
sage: maxima.cputime(t) # output random
0.568913
```

de_solve(de, vars, ics=None)

Solves a 1st or 2nd order ordinary differential equation (ODE) in two variables, possibly with initial conditions.

INPUT:

- **de** - a string representing the ODE
- **vars** - a list of strings representing the two variables.
- **ics** - a triple of numbers [a,b1,b2] representing $y(a)=b1, y'(a)=b2$

EXAMPLES:

```
sage: maxima.de_solve('diff(y,x,2) + 3*x = y', ['x','y'], [1,1,1])
y = 3*x-2*e^(x-1)
sage: maxima.de_solve('diff(y,x,2) + 3*x = y', ['x','y'])
y = %k1*e^x+%k2*e^-x+3*x
sage: maxima.de_solve('diff(y,x) + 3*x = y', ['x','y'])
y = (%c-3*((-x)-1)*e^-x)*e^x
sage: maxima.de_solve('diff(y,x) + 3*x = y', ['x','y'],[1,1])
y = -e^-1*(5*e^x-3*e*x-3*e)
```

de_solve_laplace(de, vars, ics=None)

Solves an ordinary differential equation (ODE) using Laplace transforms.

INPUT:

- **de** - a string representing the ODE (e.g., **de** = "diff(f(x),x,2)=diff(f(x),x)+sin(x)")
- **vars** - a list of strings representing the variables (e.g., vars = ["x","f"])
- **ics** - a list of numbers representing initial conditions, with symbols allowed which are represented by strings (eg, f(0)=1, f'(0)=2 is ics = [0,1,2])

EXAMPLES:

```
sage: maxima.clear('x'); maxima.clear('f')
sage: maxima.de_solve_laplace("diff(f(x),x,2) = 2*diff(f(x),x)-f(x)", ["x","f"],
↪ [0,1,2])
f(x) = x*e^x+e^x
```

```
sage: maxima.clear('x'); maxima.clear('f')
sage: f = maxima.de_solve_laplace("diff(f(x),x,2) = 2*diff(f(x),x)-f(x)", ["x",
↪ "f"])
sage: f
f(x) = x*e^x*(at(diff(f(x),x,1),x = 0))-f(0)*x*e^x+f(0)*e^x
sage: print(f)
                !
                x d      !
f(x) = x %e  ( -- (f(x))! ) - f(0) x %e  + f(0) %e
                x      x
```

(continues on next page)

(continued from previous page)

```
dx      !
        !x = 0
```

Note: The second equation sets the values of $f(0)$ and $f'(0)$ in Maxima, so subsequent ODEs involving these variables will have these initial conditions automatically imposed.

demo(*s*)

Run Maxima's demo for *s*.

INPUT:

- *s* - string

OUTPUT: none

EXAMPLES:

```
sage: maxima.demo('cf') # not tested
read and interpret file: ../share/maxima/5.34.1/demo/cf.dem

At the '_' prompt, type ';' and <enter> to get next demonstration.
frac1:cf([1,2,3,4])
...
```

describe(*s*)

Return Maxima's help for *s*.

INPUT:

- *s* - string

OUTPUT:

Maxima's help for *s*

EXAMPLES:

```
sage: maxima.help('gcd')
-- Function: gcd (<p_1>, <p_2>, <x_1>, ...)
...
```

example(*s*)

Return Maxima's examples for *s*.

INPUT:

- *s* - string

OUTPUT:

Maxima's examples for *s*

EXAMPLES:

```
sage: maxima.example('arrays')
a[n] := n*a[n-1]

          a := n a
          n   n - 1
```

(continues on next page)

(continued from previous page)

```

a[0]:1
a[5]
120
a[n]:=n
a[6]
6
a[4]
24
done

```

function(args, defn, rep=None, latex=None)

Return the Maxima function with given arguments and definition.

INPUT:

- **args** - a string with variable names separated by commas
- **defn** - a string (or Maxima expression) that defines a function of the arguments in Maxima.
- **rep** - an optional string; if given, this is how the function will print.

OUTPUT: Maxima function

EXAMPLES:

```

sage: f = maxima.function('x', 'sin(x)')
sage: f(3.2) # abs tol 2e-16
-0.058374143427579909
sage: f = maxima.function('x,y', 'sin(x)+cos(y)')
sage: f(2, 3.5) # abs tol 2e-16
sin(2)-0.9364566872907963
sage: f
sin(x)+cos(y)

```

```

sage: g = f.integrate('z')
sage: g
(cos(y)+sin(x))*z
sage: g(1,2,3)
3*(cos(2)+sin(1))

```

The function definition can be a Maxima object:

```

sage: an_expr = maxima('sin(x)*gamma(x)')
sage: t = maxima.function('x', an_expr)
sage: t
gamma(x)*sin(x)
sage: t(2)
sin(2)
sage: float(t(2))
0.9092974268256817
sage: loads(t.dumps())
gamma(x)*sin(x)

```

help(s)

Return Maxima's help for s.

INPUT:

- s - string

OUTPUT:

Maxima's help for s

EXAMPLES:

```
sage: maxima.help('gcd')
-- Function: gcd (<p_1>, <p_2>, <x_1>, ...)
...
```

plot2d(*args)

Plot a 2d graph using Maxima / gnuplot.

maxima.plot2d(f, '[var, min, max]', options)

INPUT:

- **f** - a string representing a function (such as `f="sin(x)"`) [var, xmin, xmax]
- **options** - an optional string representing plot2d options in gnuplot format

EXAMPLES:

```
sage: maxima.plot2d('sin(x)', '[x,-5,5]') # not tested
sage: opts = '[gnuplot_term, ps], [gnuplot_out_file, "sin-plot.eps"]'
sage: maxima.plot2d('sin(x)', '[x,-5,5]', opts) # not tested
```

The eps file is saved in the current directory.

plot2d_parametric(r, var, trange, nticks=50, options=None)

Plot $r = [x(t), y(t)]$ for $t = tmin...tmax$ using gnuplot with options.

INPUT:

- **r** - a string representing a function (such as `r="[x(t),y(t)]"`)
- **var** - a string representing the variable (such as `var="t"`)
- **trange** - [tmin, tmax] are numbers with tmin<tmax
- **nticks** - int (default: 50)
- **options** - an optional string representing plot2d options in gnuplot format

EXAMPLES:

```
sage: maxima.plot2d_parametric(["sin(t)", "cos(t)", "t", [-3.1, 3.1]) # not
↳tested
```

```
sage: opts = '[gnuplot_preamble, "set nokey"], [gnuplot_term, ps], [gnuplot_out_
↳file, "circle-plot.eps"]'
sage: maxima.plot2d_parametric(["sin(t)", "cos(t)", "t", [-3.1, 3.1]),
↳options=opts) # not tested
```

The eps file is saved to the current working directory.

Here is another fun plot:

```

sage: maxima.plot2d_parametric(["sin(5*t)","cos(11*t)", "t", [0,2*pi()]),
↳ nticks=400) # not tested

```

`plot3d(*args)`

Plot a 3d graph using Maxima / gnuplot.

```
maxima.plot3d(f, '[x, xmin, xmax]', '[y, ymin, ymax]', '[grid, nx, ny]', options)
```

INPUT:

- **f** - a string representing a function (such as `f="sin(x)"`) [var, min, max]
- **args** should be of the form `'[x, xmin, xmax]', '[y, ymin, ymax]', '[grid, nx, ny]', options`

EXAMPLES:

```

sage: maxima.plot3d('1 + x^3 - y^2', '[x,-2,2]', '[y,-2,2]', '[grid,12,12]')
↳ # not tested
sage: maxima.plot3d('sin(x)*cos(y)', '[x,-2,2]', '[y,-2,2]', '[grid,30,30]')
↳ # not tested
sage: opts = '[gnuplot_term, ps], [gnuplot_out_file, "sin-plot.eps"]'
sage: maxima.plot3d('sin(x+y)', '[x,-5,5]', '[y,-1,1]', opts) # not tested

```

The eps file is saved in the current working directory.

`plot3d_parametric(r, vars, urange, vrangle, options=None)`

Plot a 3d parametric graph with $r=(x,y,z)$, $x = x(u,v)$, $y = y(u,v)$, $z = z(u,v)$, for $u = \text{umin} \dots \text{umax}$, $v = \text{vmin} \dots \text{vmax}$ using gnuplot with options.

INPUT:

- **x, y, z** - a string representing a function (such as `x="u2+v2", ...`) **vars** is a list or two strings representing variables (such as `vars = ["u","v"]`)
- **urange** - [umin, umax]
- **vrangle** - [vmin, vmax] are lists of numbers with `umin umax, vmin vmax`
- **options** - optional string representing `plot2d` options in gnuplot format

OUTPUT: displays a plot on screen or saves to a file

EXAMPLES:

```

sage: maxima.plot3d_parametric(["v*sin(u)", "v*cos(u)", "v"], ["u", "v"], [-3.2, 3.
↳ 2], [0, 3]) # not tested
sage: opts = '[gnuplot_term, ps], [gnuplot_out_file, "sin-cos-plot.eps"]'
sage: maxima.plot3d_parametric(["v*sin(u)", "v*cos(u)", "v"], ["u", "v"], [-3.2, 3.
↳ 2], [0, 3], opts) # not tested

```

The eps file is saved in the current working directory.

Here is a torus:

```

sage: _ = maxima.eval("expr_1: cos(y)*(10.0+6*cos(x)); expr_2: sin(y)*(10.
↳ 0+6*cos(x)); expr_3: -6*sin(x);")
sage: maxima.plot3d_parametric(["expr_1", "expr_2", "expr_3"], ["x", "y"], [0, 6], [0,
↳ 6]) # not tested

```

Here is a Möbius strip:

```

sage: x = "cos(u)*(3 + v*cos(u/2))"
sage: y = "sin(u)*(3 + v*cos(u/2))"
sage: z = "v*sin(u/2)"
sage: maxima.plot3d_parametric([x,y,z], ["u","v"], [-3.1,3.2], [-1/10,1/10]) #
↳not tested

```

plot_list(*ptsx, ptsy, options=None*)

Plots a curve determined by a sequence of points.

INPUT:

- **ptsx** - [x1,...,xn], where the xi and yi are real,
- **ptsy** - [y1,...,yn]
- **options** - a string representing maxima plot2d options.

The points are (x1,y1), (x2,y2), etc.

This function requires maxima 5.9.2 or newer.

Note: More than 150 points can sometimes lead to the program hanging. Why?

EXAMPLES:

```

sage: zeta_ptsx = [ (pari(1/2 + i*I/10).zeta().real()).precision(1) for i in
↳range(70,150)]
sage: zeta_ptsy = [ (pari(1/2 + i*I/10).zeta().imag()).precision(1) for i in
↳range(70,150)]
sage: maxima.plot_list(zeta_ptsx, zeta_ptsy) # not tested
sage: opts='[gnuplot_preamble, "set nokey"], [gnuplot_term, ps], [gnuplot_out_
↳file, "zeta.eps"]'
sage: maxima.plot_list(zeta_ptsx, zeta_ptsy, opts) # not tested

```

plot_multilist(*pts_list, options=None*)

Plots a list of list of points pts_list=[pts1,pts2,...,ptsn], where each ptsi is of the form [[x1,y1],...,[xn,yn]] x's must be integers and y's reals options is a string representing maxima plot2d options.

INPUT:

- **pts_lst** - list of points; each point must be of the form [x,y] where x is an integer and y is a real
- **var** - string; representing Maxima's plot2d options

Requires maxima 5.9.2 at least.

Note: More than 150 points can sometimes lead to the program hanging.

EXAMPLES:

```

sage: xx = [ i/10.0 for i in range(-10,10)]
sage: yy = [ i/10.0 for i in range(-10,10)]
sage: x0 = [ 0 for i in range(-10,10)]
sage: y0 = [ 0 for i in range(-10,10)]
sage: zeta_ptsx1 = [ (pari(1/2+i*I/10).zeta().real()).precision(1) for i in
↳range(10)]

```

(continues on next page)

(continued from previous page)

```

sage: zeta_ptsy1 = [ (pari(1/2+i*I/10).zeta().imag()).precision(1) for i in
↳range(10)]
sage: maxima.plot_multilist([[zeta_ptsx1,zeta_ptsy1],[xx,y0],[x0,yy]])          #
↳not tested
sage: zeta_ptsx1 = [ (pari(1/2+i*I/10).zeta().real()).precision(1) for i in
↳range(10,150)]
sage: zeta_ptsy1 = [ (pari(1/2+i*I/10).zeta().imag()).precision(1) for i in
↳range(10,150)]
sage: maxima.plot_multilist([[zeta_ptsx1,zeta_ptsy1],[xx,y0],[x0,yy]])          #
↳not tested
sage: opts='[gnuplot_preamble, "set nokey"]'
sage: maxima.plot_multilist([[zeta_ptsx1,zeta_ptsy1],[xx,y0],[x0,yy]],opts)
↳# not tested

```

solve_linear(eqns, vars)

Wraps maxima's linsolve.

INPUT:

- eqns - a list of m strings; each representing a linear question in $m = n$ variables
- vars - a list of n strings; each representing a variable

EXAMPLES:

```

sage: eqns = ["x + z = y", "2*a*x - y = 2*a^2", "y - 2*z = 2"]
sage: vars = ["x", "y", "z"]
sage: maxima.solve_linear(eqns, vars)
[x = a+1,y = 2*a,z = a-1]

```

unit_quadratic_integer(n)Finds a unit of the ring of integers of the quadratic number field $\mathbf{Q}(\sqrt{n})$, $n > 1$, using the qunit maxima command.

INPUT:

- n - an integer

EXAMPLES:

```

sage: u = maxima.unit_quadratic_integer(101); u
a + 10
sage: u.parent()
Number Field in a with defining polynomial x^2 - 101 with a = 10.04987562112089?
sage: u = maxima.unit_quadratic_integer(13)
sage: u
5*a + 18
sage: u.parent()
Number Field in a with defining polynomial x^2 - 13 with a = 3.605551275463990?

```

version()

Return the version of Maxima that Sage includes.

INPUT: none

OUTPUT: none

EXAMPLES:

```
sage: maxima.version() # random
'5.41.0'
```

class `sage.interfaces.maxima_abstract.MaximaAbstractElement`(*parent, value, is_name=False, name=None*)

Bases: `sage.interfaces.tab_completion.ExtraTabCompletion`, `sage.interfaces.interface.InterfaceElement`

Element of Maxima through an abstract interface.

EXAMPLES:

Elements of this class should not be created directly. The targeted parent of a concrete inherited class should be used instead:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: xp = maxima(x)
sage: type(xp)
<class 'sage.interfaces.maxima.MaximaElement'>
sage: xl = maxima_lib(x)
sage: type(xl)
<class 'sage.interfaces.maxima_lib.MaximaLibElement'>
```

comma(*args*)

Form the expression that would be written 'self, args' in Maxima.

INPUT:

- *args* - string

OUTPUT: Maxima object

EXAMPLES:

```
sage: maxima('sqrt(2) + I').comma('numer')
I+1.41421356237309...
sage: maxima('sqrt(2) + I*a').comma('a=5')
5*I+sqrt(2)
```

derivative(*var='x', n=1*)

Return the n-th derivative of self.

INPUT:

- *var* - variable (default: 'x')
- *n* - integer (default: 1)

OUTPUT: n-th derivative of self with respect to the variable *var*

EXAMPLES:

```
sage: f = maxima('x^2')
sage: f.diff()
2*x
sage: f.diff('x')
2*x
sage: f.diff('x', 2)
2
```

(continues on next page)

(continued from previous page)

```
sage: maxima('sin(x^2)').diff('x',4)
16*x^4*sin(x^2)-12*sin(x^2)-48*x^2*cos(x^2)
```

```
sage: f = maxima('x^2 + 17*y^2')
sage: f.diff('x')
34*y*'diff(y,x,1)+2*x
sage: f.diff('y')
34*y
```

diff(*var='x', n=1*)

Return the n-th derivative of self.

INPUT:

- *var* - variable (default: 'x')
- *n* - integer (default: 1)

OUTPUT: n-th derivative of self with respect to the variable *var*

EXAMPLES:

```
sage: f = maxima('x^2')
sage: f.diff()
2*x
sage: f.diff('x')
2*x
sage: f.diff('x', 2)
2
sage: maxima('sin(x^2)').diff('x',4)
16*x^4*sin(x^2)-12*sin(x^2)-48*x^2*cos(x^2)
```

```
sage: f = maxima('x^2 + 17*y^2')
sage: f.diff('x')
34*y*'diff(y,x,1)+2*x
sage: f.diff('y')
34*y
```

dot(*other*)Implements the notation *self* . *other*.

INPUT:

- *other* - matrix; argument to dot.

OUTPUT: Maxima matrix

EXAMPLES:

```
sage: A = maxima('matrix ([a1],[a2])')
sage: B = maxima('matrix ([b1, b2])')
sage: A.dot(B)
matrix([a1*b1, a1*b2], [a2*b1, a2*b2])
```

imag()

Return the imaginary part of this Maxima element.

INPUT: none

OUTPUT: Maxima real

EXAMPLES:

```
sage: maxima('2 + (2/3)*%i').imag()
2/3
```

integral(*var='x', min=None, max=None*)

Return the integral of self with respect to the variable x.

INPUT:

- var - variable
- min - default: None
- max - default: None

OUTPUT:

- the definite integral if xmin is not None
- an indefinite integral otherwise

EXAMPLES:

```
sage: maxima('x^2+1').integral()
x^3/3+x
sage: maxima('x^2+ 1 + y^2').integral('y')
y^3/3+x^2*y+y
sage: maxima('x / (x^2+1)').integral()
log(x^2+1)/2
sage: maxima('1/(x^2+1)').integral()
atan(x)
sage: maxima('1/(x^2+1)').integral('x', 0, infinity)
%pi/2
sage: maxima('x/(x^2+1)').integral('x', -1, 1)
0
```

```
sage: f = maxima('exp(x^2)').integral('x',0,1); f
-(sqrt(%pi)*%i*erf(%i))/2
sage: f.numer()
1.46265174590718...
```

integrate(*var='x', min=None, max=None*)

Return the integral of self with respect to the variable x.

INPUT:

- var - variable
- min - default: None
- max - default: None

OUTPUT:

- the definite integral if xmin is not None
- an indefinite integral otherwise

EXAMPLES:

```

sage: maxima('x^2+1').integral()
x^3/3+x
sage: maxima('x^2+ 1 + y^2').integral('y')
y^3/3+x^2*y+y
sage: maxima('x / (x^2+1)').integral()
log(x^2+1)/2
sage: maxima('1/(x^2+1)').integral()
atan(x)
sage: maxima('1/(x^2+1)').integral('x', 0, infinity)
%pi/2
sage: maxima('x/(x^2+1)').integral('x', -1, 1)
0

```

```

sage: f = maxima('exp(x^2)').integral('x',0,1); f
-(sqrt(%pi)*%i*erf(%i))/2
sage: f.numer()
1.46265174590718...

```

nintegral(var='x', a=0, b=1, desired_relative_error='1e-8', maximum_num_subintervals=200)

Return a numerical approximation to the integral of self from a to b.

INPUT:

- var - variable to integrate with respect to
- a - lower endpoint of integration
- b - upper endpoint of integration
- **desired_relative_error** - (default: '1e-8') the desired relative error
- **maximum_num_subintervals** - (default: 200) maxima number of subintervals

OUTPUT:

- approximation to the integral
- **estimated absolute error of the** approximation
- the number of integrand evaluations
- an error code:
 - 0 - no problems were encountered
 - 1 - too many subintervals were done
 - 2 - excessive roundoff error
 - 3 - extremely bad integrand behavior
 - 4 - failed to converge
 - 5 - integral is probably divergent or slowly convergent
 - 6 - the input is invalid

EXAMPLES:

```

sage: maxima('exp(-sqrt(x))').nintegral('x',0,1)
(0.5284822353142306, 4.1633141378838...e-11, 231, 0)

```

Note that GP also does numerical integration, and can do so to very high precision very quickly:

```
sage: gp('intnum(x=0,1,exp(-sqrt(x)))')
0.5284822353142307136179049194 # 32-bit
0.52848223531423071361790491935415653022 # 64-bit
sage: _ = gp.set_precision(80)
sage: gp('intnum(x=0,1,exp(-sqrt(x)))')
0.
↪52848223531423071361790491935415653021675547587292866196865279321015401702040079
```

numer()

Return numerical approximation to self as a Maxima object.

INPUT: none

OUTPUT: Maxima object

EXAMPLES:

```
sage: a = maxima('sqrt(2)').numer(); a
1.41421356237309...
sage: type(a)
<class 'sage.interfaces.maxima.MaximaElement'>
```

partial_fraction_decomposition(var='x')

Return the partial fraction decomposition of self with respect to the variable var.

INPUT:

- var - string

OUTPUT: Maxima object

EXAMPLES:

```
sage: f = maxima('1/((1+x)*(x-1))')
sage: f.partial_fraction_decomposition('x')
1/(2*(x-1))-1/(2*(x+1))
sage: print(f.partial_fraction_decomposition('x'))
          1           1
-----  -  -----
      2 (x - 1)    2 (x + 1)
```

real()

Return the real part of this Maxima element.

INPUT: none

OUTPUT: Maxima real

EXAMPLES:

```
sage: maxima('2 + (2/3)*%i').real()
2
```

str()

Return string representation of this Maxima object.

INPUT: none

OUTPUT: string

EXAMPLES:

```
sage: maxima('sqrt(2) + 1/3').str()
'sqrt(2)+1/3'
```

subst(*val*)

Substitute a value or several values into this Maxima object.

INPUT:

- *val* - string representing substitution(s) to perform

OUTPUT: Maxima object

EXAMPLES:

```
sage: maxima('a^2 + 3*a + b').subst('b=2')
a^2+3*a+2
sage: maxima('a^2 + 3*a + b').subst('a=17')
b+340
sage: maxima('a^2 + 3*a + b').subst('a=17, b=2')
342
```

class `sage.interfaces.maxima_abstract.MaximaAbstractElementFunction`(*parent, name, defn, args, latex*)

Bases: `sage.interfaces.maxima_abstract.MaximaAbstractElement`

Create a Maxima function with the parent *parent*, name *name*, definition *defn*, arguments *args* and latex representation *latex*.

INPUT:

- *parent* - an instance of a concrete Maxima interface
- *name* - string
- *defn* - string
- *args* - string; comma separated names of arguments
- *latex* - string

OUTPUT: Maxima function

EXAMPLES:

```
sage: maxima.function('x,y','e^cos(x)')
e^cos(x)
```

arguments(*split=True*)

Returns the arguments of this Maxima function.

INPUT:

- *split* - boolean; if True return a tuple of strings, otherwise return a string of comma-separated arguments

OUTPUT:

- a string if *split* is False
- a list of strings if *split* is True

EXAMPLES:

```
sage: f = maxima.function('x,y', 'sin(x+y)')
sage: f.arguments()
['x', 'y']
sage: f.arguments(split=False)
'x,y'
sage: f = maxima.function('', 'sin(x)')
sage: f.arguments()
[]
```

definition()

Returns the definition of this Maxima function as a string.

INPUT: none

OUTPUT: string

EXAMPLES:

```
sage: f = maxima.function('x,y', 'sin(x+y)')
sage: f.definition()
'sin(x+y)'
```

integral(*var*)

Returns the integral of self with respect to the variable *var*.

INPUT:

- *var* - a variable

OUTPUT: Maxima function

Note that `integrate` is an alias of `integral`.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f = maxima.function('x', 'sin(x)')
sage: f.integral(x)
-cos(x)
sage: f.integral(y)
sin(x)*y
```

integrate(*var*)

Returns the integral of self with respect to the variable *var*.

INPUT:

- *var* - a variable

OUTPUT: Maxima function

Note that `integrate` is an alias of `integral`.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f = maxima.function('x', 'sin(x)')
sage: f.integral(x)
-cos(x)
```

(continues on next page)

(continued from previous page)

```
sage: f.integral(y)
sin(x)*y
```

`sage.interfaces.maxima_abstract.maxima_console()`
Spawn a new Maxima command-line session.

EXAMPLES:

```
sage: from sage.interfaces.maxima_abstract import maxima_console
sage: maxima_console() # not tested
Maxima 5.34.1 http://maxima.sourceforge.net
...
```

`sage.interfaces.maxima_abstract.maxima_version()`
Return Maxima version.

Currently this calls a new copy of Maxima.

EXAMPLES:

```
sage: from sage.interfaces.maxima_abstract import maxima_version
sage: maxima_version() # random
'5.41.0'
```

`sage.interfaces.maxima_abstract.reduce_load_MaximaAbstract_function(parent, defn, args, latex)`
Unpickle a Maxima function.

EXAMPLES:

```
sage: from sage.interfaces.maxima_abstract import reduce_load_MaximaAbstract_
↪function
sage: f = maxima.function('x,y','sin(x+y)')
sage: _,args = f.__reduce__()
sage: g = reduce_load_MaximaAbstract_function(*args)
sage: g == f
True
```


LIBRARY INTERFACE TO MAXIMA

Maxima is a free GPL'd general purpose computer algebra system whose development started in 1968 at MIT. It contains symbolic manipulation algorithms, as well as implementations of special functions, including elliptic functions and generalized hypergeometric functions. Moreover, Maxima has implementations of many functions relating to the invariant theory of the symmetric group S_n . (However, the commands for group invariants, and the corresponding Maxima documentation, are in French.) For many links to Maxima documentation, see <http://maxima.sourceforge.net/documentation.html>.

AUTHORS:

- William Stein (2005-12): Initial version
- David Joyner: Improved documentation
- William Stein (2006-01-08): Fixed bug in parsing
- William Stein (2006-02-22): comparisons (following suggestion of David Joyner)
- William Stein (2006-02-24): *greatly* improved robustness by adding sequence numbers to IO bracketing in `_eval_line`
- Robert Bradshaw, Nils Bruin, Jean-Pierre Flori (2010,2011): Binary library interface

For this interface, Maxima is loaded into ECL which is itself loaded as a C library in Sage. Translations between Sage and Maxima objects (which are nothing but wrappers to ECL objects) is made as much as possible directly, but falls back to the string based conversion used by the classical Maxima Pexpect interface in case no new implementation has been made.

This interface is the one used for calculus by Sage and is accessible as *maxima_calculus*:

```
sage: maxima_calculus
Maxima_lib
```

Only one instance of this interface can be instantiated, so the user should not try to instantiate another one, which is anyway set to raise an error:

```
sage: from sage.interfaces.maxima_lib import MaximaLib
sage: MaximaLib()
Traceback (most recent call last):
...
RuntimeError: Maxima interface in library mode can only be instantiated once
```

Changed `besselexpand` to true in `init_code` – automatically simplify Bessel functions to trig functions when appropriate when true. Examples:

For some infinite sums, a closed expression can be found. By default, “maxima” is used for that:

```
sage: x,n,k = var("x","n","k")
sage: sum((-x)^n/(factorial(n)*factorial(n+3/2)),n,0,oo)
-1/2*(2*x*cos(2*sqrt(x)) - sqrt(x)*sin(2*sqrt(x)))/(sqrt(pi)*x^2)
```

Maxima has some flags that affect how the result gets simplified (By default, `besselexpand` is false in Maxima; however in 5.39 this test does not show any difference, as, apparently, another expansion path is used):

```
sage: maxima_calculus("besselexpand:false")
false
sage: x,n,k = var("x","n","k")
sage: sum((-x)^n/(factorial(n)*factorial(n+3/2)),n,0,oo)
-1/2*(2*x*cos(2*sqrt(x)) - sqrt(x)*sin(2*sqrt(x)))/(sqrt(pi)*x^2)
sage: maxima_calculus("besselexpand:true")
true
```

The output is parseable (i. e. [trac ticket #31796](#) is fixed):

```
sage: foo = maxima_calculus('a and (b or c)') ; foo
a and (b or c)
sage: bar = maxima_calculus(foo) ; bar
a and (b or c)
sage: bar == foo
True
```

class `sage.interfaces.maxima_lib.Maximalib`

Bases: `sage.interfaces.maxima_abstract.MaximaAbstract`

Interface to Maxima as a Library.

INPUT: none

OUTPUT: Maxima interface as a Library

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import Maximalib, maxima_lib
sage: isinstance(maxima_lib,Maximalib)
True
```

Only one such interface can be instantiated:

```
sage: Maximalib()
Traceback (most recent call last):
...
RuntimeError: Maxima interface in library mode can only
be instantiated once
```

clear(*var*)

Clear the variable named *var*.

INPUT:

- *var* - string

OUTPUT: none

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib.set('xxxxx', '2')
sage: maxima_lib.get('xxxxx')
'2'
sage: maxima_lib.clear('xxxxx')
sage: maxima_lib.get('xxxxx')
'xxxxx'
```

eval(*line*, *locals=None*, *reformat=True*, ***kwds*)

Evaluate the line in Maxima.

INPUT:

- *line* - string; text to evaluate
- *locals* - None (ignored); this is used for compatibility with the Sage notebook's generic system interface.
- *reformat* - boolean; whether to strip output or not
- ***kwds* - All other arguments are currently ignored.

OUTPUT: string representing Maxima output

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib._eval_line('1+1')
'2'
sage: maxima_lib._eval_line('1+1;')
'2'
sage: maxima_lib._eval_line('1+1$')
''
sage: maxima_lib._eval_line('randvar : cos(x)+sin(y)$')
''
sage: maxima_lib._eval_line('randvar')
'sin(y)+cos(x)'
```

get(*var*)

Get the string value of the variable *var*.

INPUT:

- *var* - string

OUTPUT: string

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib.set('xxxxx', '2')
sage: maxima_lib.get('xxxxx')
'2'
```

lisp(*cmd*)

Send a lisp command to maxima.

INPUT:

- *cmd* - string

OUTPUT: ECL object

Note: The output of this command is very raw - not pretty.

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib.lisp("(+ 2 17)")
<ECL: 19>
```

set(*var, value*)

Set the variable *var* to the given value.

INPUT:

- *var* - string
- *value* - string

OUTPUT: none

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib.set('xxxxx', '2')
sage: maxima_lib.get('xxxxx')
'2'
```

sr_integral(**args*)

Helper function to wrap calculus use of Maxima's integration.

sr_limit(*expr, v, a, dir=None*)

Helper function to wrap calculus use of Maxima's limits.

sr_prod(**args*)

Helper function to wrap calculus use of Maxima's product.

sr_sum(**args*)

Helper function to wrap calculus use of Maxima's summation.

sr_tlimit(*expr, v, a, dir=None*)

Helper function to wrap calculus use of Maxima's Taylor series limits.

class `sage.interfaces.maxima_lib.MaximaLibElement`(*parent, value, is_name=False, name=None*)

Bases: `sage.interfaces.maxima_abstract.MaximaAbstractElement`

Element of Maxima through library interface.

EXAMPLES:

Elements of this class should not be created directly. The targeted parent should be used instead:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib(4)
4
sage: maxima_lib(log(x))
log(_SAGE_VAR_x)
```

display2d(*onscreen=True*)

Return the 2d representation of this Maxima object.

INPUT:

- `onscreen` - boolean (default: True); whether to print or return

OUTPUT:

The representation is printed if `onscreen` is set to True and returned as a string otherwise.

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: F = maxima_lib('x^5 - y^5').factor()
sage: F.display2d()
      4      3      2 2      3      4
    - (y - x) (y + x y + x y + x y + x )
```

`ecl()`

Return the underlying ECL object of this MaximaLib object.

INPUT: none

OUTPUT: ECL object

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: maxima_lib(x+cos(19)).ecl()
<ECL: ((MPLUS SIMP) ((%COS SIMP) 19) |$_SAGE_VAR_x|)>
```

`to_poly_solve(vars, options="")`

Use Maxima's `to_poly_solver` package.

INPUT:

- `vars` - symbolic expressions
- `options` - string (default="")

OUTPUT: Maxima object

EXAMPLES:

The `zXXX` below are names for arbitrary integers and subject to change:

```
sage: from sage.interfaces.maxima_lib import maxima_lib
sage: sol = maxima_lib(sin(x) == 0).to_poly_solve(x)
sage: sol.sage()
[[x == pi*z...]]
```

`class sage.interfaces.maxima_lib.MaximaLibElementFunction(parent, name, defn, args, latex)`

Bases: `sage.interfaces.maxima_lib.MaximaLibElement`, `sage.interfaces.maxima_abstract.MaximaAbstractElementFunction`

`sage.interfaces.maxima_lib.dummy_integrate(expr)`

We would like to simply tie Maxima's `integrate` to `sage.calculus.calculus.dummy_integrate`, but we're being imported there so to avoid circularity we define it here.

INPUT:

- `expr` - ECL object; a Maxima `%INTEGRATE` expression

OUTPUT: symbolic expression

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib, dummy_integrate
sage: f = maxima_lib('f(x)').integrate('x')
sage: f.ecl()
<ECL: ((%INTEGRATE SIMP) ((\$F SIMP) \$X) \$X)>
sage: dummy_integrate(f.ecl())
integrate(f(x), x)
```

```
sage: f = maxima_lib('f(x)').integrate('x',0,10)
sage: f.ecl()
<ECL: ((%INTEGRATE SIMP) ((\$F SIMP) \$X) \$X 0 10)>
sage: dummy_integrate(f.ecl())
integrate(f(x), x, 0, 10)
```

`sage.interfaces.maxima_lib.is_MaximaLibElement(x)`
Returns True if x is of type MaximaLibElement.

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib, is_MaximaLibElement
sage: m = maxima_lib(1)
sage: is_MaximaLibElement(m)
True
sage: is_MaximaLibElement(1)
False
```

`sage.interfaces.maxima_lib.max_at_to_sage(expr)`
Special conversion rule for AT expressions.

INPUT:

- `expr` - ECL object; a Maxima AT expression

OUTPUT: symbolic expression

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib, max_at_to_sage
sage: a=maxima_lib("'at(f(x,y,z),[x=1,y=2,z=3])")
sage: a
'at(f(x,y,z),[x = 1,y = 2,z = 3])
sage: max_at_to_sage(a.ecl())
f(1, 2, 3)
sage: a=maxima_lib("'at(f(x,y,z),x=1)")
sage: a
'at(f(x,y,z),x = 1)
sage: max_at_to_sage(a.ecl())
f(1, y, z)
```

`sage.interfaces.maxima_lib.max_harmonic_to_sage(expr)`

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib, max_to_sr
sage: c=maxima_lib(harmonic_number(x,2))
sage: c.ecl()
<ECL: ((\$GEN_HARMONIC_NUMBER SIMP) 2 |$_SAGE_VAR_x|)>
```

(continues on next page)

(continued from previous page)

```
sage: max_to_sr(c.ecl())
harmonic_number(x, 2)
```

sage.interfaces.maxima_lib.max_pochhammer_to_sage(*expr*)

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib, max_to_sr
sage: c = maxima_lib('pochhammer(x,n)')
sage: c.ecl()
<ECL: (($POCHHAMMER SIMP) $X $N)>
sage: max_to_sr(c.ecl())
gamma(n + x)/gamma(x)
```

sage.interfaces.maxima_lib.max_to_sr(*expr*)

Convert a Maxima object into a symbolic expression.

INPUT:

- *expr* - ECL object

OUTPUT: symbolic expression

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib, max_to_sr
sage: f = maxima_lib('f(x)')
sage: f.ecl()
<ECL: (($F SIMP) $X)>
sage: max_to_sr(f.ecl())
f(x)
```

sage.interfaces.maxima_lib.max_to_string(*s*)

Return the Maxima string corresponding to this ECL object.

INPUT:

- *s* - ECL object

OUTPUT: string

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib, max_to_string
sage: ecl = maxima_lib(cos(x)).ecl()
sage: max_to_string(ecl)
'cos(_SAGE_VAR_x)'
```

sage.interfaces.maxima_lib.mdiff_to_sage(*expr*)

Special conversion rule for %DERIVATIVE expressions.

INPUT:

- *expr* - ECL object; a Maxima %DERIVATIVE expression

OUTPUT: symbolic expression

EXAMPLES:

```

sage: from sage.interfaces.maxima_lib import maxima_lib, mdiff_to_sage
sage: f = maxima_lib('f(x)').diff('x',4)
sage: f.ecl()
<ECL: ((%DERIVATIVE SIMP) (($F SIMP) $X) $X 4)>
sage: mdiff_to_sage(f.ecl())
diff(f(x), x, x, x, x)

```

`sage.interfaces.maxima_lib.mlist_to_sage(expr)`

Special conversion rule for MLIST expressions.

INPUT:

- *expr* - ECL object; a Maxima MLIST expression (i.e., a list)

OUTPUT: a Python list of converted expressions.

EXAMPLES:

```

sage: from sage.interfaces.maxima_lib import maxima_lib, mlist_to_sage
sage: L=maxima_lib("[1,2,3]")
sage: L.ecl()
<ECL: ((MLIST SIMP) 1 2 3)>
sage: mlist_to_sage(L.ecl())
[1, 2, 3]

```

`sage.interfaces.maxima_lib.mqapply_to_sage(expr)`

Special conversion rule for MQAPPLY expressions.

INPUT:

- *expr* - ECL object; a Maxima MQAPPLY expression

OUTPUT: symbolic expression

MQAPPLY is used for function as $\text{li}[x](y)$ and $\text{psi}[x](y)$.

EXAMPLES:

```

sage: from sage.interfaces.maxima_lib import maxima_lib, mqapply_to_sage
sage: c = maxima_lib('li[2](3)')
sage: c.ecl()
<ECL: ((MQAPPLY SIMP) (($LI SIMP ARRAY) 2) 3)>
sage: mqapply_to_sage(c.ecl())
dilog(3)

```

`sage.interfaces.maxima_lib.mrat_to_sage(expr)`

Convert a Maxima MRAT expression to Sage SR.

INPUT:

- *expr* - ECL object; a Maxima MRAT expression

OUTPUT: symbolic expression

Maxima has an optimised representation for multivariate rational expressions. The easiest way to translate those to SR is by first asking Maxima to give the generic representation of the object. That is what RATDISREP does in Maxima.

EXAMPLES:

```

sage: from sage.interfaces.maxima_lib import maxima_lib, mrat_to_sage
sage: var('x y z')
(x, y, z)
sage: c = maxima_lib((x+y^2+z^9)/x^6+z^8/y).rat()
sage: c
(_SAGE_VAR_y*_SAGE_VAR_z^9+_SAGE_VAR_x^6*_SAGE_VAR_z^8+_SAGE_VAR_y^3+_SAGE_VAR_x*_
↪SAGE_VAR_y)/(_SAGE_VAR_x^6*_SAGE_VAR_y)
sage: c.ecl()
<ECL: ((MRAT SIMP (|$_SAGE_VAR_x| |$_SAGE_VAR_y| |$_SAGE_VAR_z|)
...>
sage: mrat_to_sage(c.ecl())
(x^6*z^8 + y*z^9 + y^3 + x*y)/(x^6*y)
    
```

`sage.interfaces.maxima_lib.parse_max_string(s)`
 Evaluate string in Maxima without *any* further simplification.

INPUT:

- s - string

OUTPUT: ECL object

EXAMPLES:

```

sage: from sage.interfaces.maxima_lib import parse_max_string
sage: parse_max_string('1+1')
<ECL: ((MPLUS) 1 1)>
    
```

`sage.interfaces.maxima_lib.pyobject_to_max(obj)`
 Convert a (simple) Python object into a Maxima object.

INPUT:

- expr - Python object

OUTPUT: ECL object

Note: This uses functions defined in `sage.libs.ecl`.

EXAMPLES:

```

sage: from sage.interfaces.maxima_lib import pyobject_to_max
sage: pyobject_to_max(4)
<ECL: 4>
sage: pyobject_to_max('z')
<ECL: Z>
sage: var('x')
x
sage: pyobject_to_max(x)
Traceback (most recent call last):
...
TypeError: Unimplemented type for python_to_ecl
    
```

`sage.interfaces.maxima_lib.reduce_load_MaximaLib()`
 Unpickle the (unique) Maxima library interface.

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import reduce_load_MaximaLib
sage: reduce_load_MaximaLib()
Maxima_lib
```

`sage.interfaces.maxima_lib.sage_rat(x, y)`

Return quotient x/y .

INPUT:

- x - integer
- y - integer

OUTPUT: rational

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import sage_rat
sage: sage_rat(1,7)
1/7
```

`sage.interfaces.maxima_lib.sr_to_max(expr)`

Convert a symbolic expression into a Maxima object.

INPUT:

- $expr$ - symbolic expression

OUTPUT: ECL object

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import sr_to_max
sage: var('x')
x
sage: sr_to_max(x)
<ECL: $X>
sage: sr_to_max(cos(x))
<ECL: ((%COS) $X)>
sage: f = function('f')(x)
sage: sr_to_max(f.diff())
<ECL: ((%DERIVATIVE) ((%F) $X) $X 1)>
```

`sage.interfaces.maxima_lib.stdout_to_string(s)`

Evaluate command s and catch Maxima stdout (not the result of the command!) into a string.

INPUT:

- s - string; command to evaluate

OUTPUT: string

This is currently used to implement `display2d()`.

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import stdout_to_string
sage: stdout_to_string('1+1')
''
sage: stdout_to_string('disp(1+1)')
'2\n\n'
```

INTERFACE TO MUPAD

AUTHOR:

- Mike Hansen
- William Stein

You must have the optional commercial MuPAD interpreter installed and available as the command code{mupkern} in your PATH in order to use this interface. You do not have to install any optional sage packages.

class `sage.interfaces.mupad.Mupad`(*maxread=None, script_subdirectory=None, server=None, server_tmpdir=None, logfile=None*)

Bases: `sage.interfaces.tab_completion.ExtraTabCompletion`, `sage.interfaces.expect.Expect`

Interface to the MuPAD interpreter.

completions(*string, strip=False*)

EXAMPLES:

```
sage: mupad.completions('linal') # optional - mupad
['linalg']
```

console()

Spawn a new MuPAD command-line session.

EXAMPLES:

```
sage: mupad.console() #not tested

*----*      MuPAD Pro 4.0.2 -- The Open Computer Algebra System
/|  /|
*----* |      Copyright (c) 1997 - 2007 by SciFace Software
| *--|-*      All rights reserved.
|/  |/
*----*      Licensed to:  ...
```

cputime(*t=None*)

EXAMPLES:

```
sage: t = mupad.cputime() #random, optional - MuPAD
0.116000000000000001
```

eval(*code, strip=True, **kwds*)

EXAMPLES:

```
sage: mupad.eval('2+2') # optional - mupad
4
```

expect()

EXAMPLES:

```
sage: a = mupad(1) # optional - mupad
sage: mupad.expect() # optional - mupad
<pexpect.spawn instance at 0x...>
```

get(var)

Get the value of the variable var.

EXAMPLES:

```
sage: mupad.set('a', 4) # optional - mupad
sage: mupad.get('a').strip() # optional - mupad
'4'
```

set(var, value)

Set the variable var to the given value.

EXAMPLES:

```
sage: mupad.set('a', 4) # optional - mupad
sage: mupad.get('a').strip() # optional - mupad
'4'
```

```
class sage.interfaces.mupad.MupadElement(parent, value, is_name=False, name=None)
Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.expect.ExpectElement
```

```
class sage.interfaces.mupad.MupadFunction(parent, name)
Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.expect.ExpectFunction
```

```
class sage.interfaces.mupad.MupadFunctionElement(obj, name)
Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.expect.FunctionElement
```

sage.interfaces.mupad.mupad_console()
Spawn a new MuPAD command-line session.

EXAMPLES:

```
sage: from sage.interfaces.mupad import mupad_console
sage: mupad_console() #not tested

*----*   MuPAD Pro 4.0.2 -- The Open Computer Algebra System
/|  /|
*----* |   Copyright (c) 1997 - 2007 by SciFace Software
| *--|-*   All rights reserved.
|/  |/
*----*   Licensed to:   ...
```

sage.interfaces.mupad.reduce_load_mupad()

EXAMPLES:

```
sage: from sage.interfaces.mupad import reduce_load_mupad
sage: reduce_load_mupad()
Mupad
```


INTERFACE TO MWRANK

`sage.interfaces.mwrank.Mwrank`(*options=""*, *server=None*, *server_tmpdir=None*)
Create and return an mwrank interpreter, with given options.

INPUT:

- *options* - string; passed when starting mwrank. The format is:

-h	help	prints this info and quits
-q	quiet	turns OFF banner display and prompt
-v n	verbosity	sets verbosity to n (default=1)
-o	PARI/GP output	turns ON extra PARI/GP short output (default is OFF)
-p n	precision	sets precision to n decimals (default=15)
-b n	quartic bound	bound on quartic point search (default=10)
-x n	n aux	number of aux primes used for sieving (default=6)
-l	list	turns ON listing of points (default ON unless v=0)
-s	selmer_only	if set, computes Selmer rank only (default: not set)
-d	skip_2nd_descent	if set, skips the second descent for curves. ↳with 2-torsion (default: not set)
-S n	sat_bd	upper bound on saturation primes (default=100, -1 for ↳automatic)

Warning: Do not use the option “-q” which turns off the prompt.

EXAMPLES:

```
sage: M = Mwrank('-v 0 -l')
sage: print(M('0 0 1 -1 0'))
Curve [0,0,1,-1,0] : Rank = 1
Generator 1 is [0:-1:1]; height 0.051...
Regulator = 0.051...
```

class `sage.interfaces.mwrank.Mwrank_class`(*options=""*, *server=None*, *server_tmpdir=None*)

Bases: `sage.interfaces.expect.Expect`

Interface to the Mwrank interpreter.

console()

Start the mwrank console.

EXAMPLES:

```
sage: mwrnk.console() # not tested: expects console input
Program mwrnk: ...
```

eval(*s*, ***kws*)

Return mwrnk's output for the given input.

INPUT:

- *s* (str) - a Sage object which when converted to a string gives valid input to mwrnk. The conversion is done by `validate_mwrnk_input()`. Possible formats are:
 - a string representing exactly five integers separated by whitespace, for example '1 2 3 4 5'
 - a string representing exactly five integers separated by commas, preceded by '[' and followed by ']' (with arbitrary whitespace), for example '[1 2 3 4 5]'
 - a list or tuple of exactly 5 integers.

Note: If a `RuntimeError` exception is raised, then the mwrnk interface is restarted and the command is retried once.

EXAMPLES:

```
sage: mwrnk.eval('12 3 4 5 6')
'Curve [12,3,4,5,6] :...'
sage: mwrnk.eval('[12, 3, 4, 5, 6]')
'Curve [12,3,4,5,6] :...'
sage: mwrnk.eval([12, 3, 4, 5, 6])
'Curve [12,3,4,5,6] :...'
sage: mwrnk.eval((12, 3, 4, 5, 6))
'Curve [12,3,4,5,6] :...'
```

`sage.interfaces.mwrnk.mwrnk_console()`

Start the mwrnk console.

EXAMPLES:

```
sage: mwrnk_console() # not tested: expects console input
Program mwrnk: ...
```

`sage.interfaces.mwrnk.validate_mwrnk_input(s)`

Returns a string suitable for mwrnk input, or raises an error.

INPUT:

- *s* – one of the following:
 - a list or tuple of 5 integers [a1,a2,a3,a4,a6] or (a1,a2,a3,a4,a6)
 - a string of the form '[a1,a2,a3,a4,a6]' or 'a1 a2 a3 a4 a6' where a1, a2, a3, a4, a6 are integers

OUTPUT:

For valid input, a string of the form '[a1,a2,a3,a4,a6]'. For invalid input a `ValueError` is raised.

EXAMPLES:

A list or tuple of 5 integers:

```

sage: from sage.interfaces.mwrank import validate_mwrank_input
sage: validate_mwrank_input([1,2,3,4,5])
'[1, 2, 3, 4, 5]'
sage: validate_mwrank_input((-1,2,-3,4,-55))
'[-1, 2, -3, 4, -55]'
sage: validate_mwrank_input([1,2,3,4])
Traceback (most recent call last):
...
ValueError: [1, 2, 3, 4] is not valid input to mwrank (should have 5 entries)
sage: validate_mwrank_input([1,2,3,4,i])
Traceback (most recent call last):
...
ValueError: [1, 2, 3, 4, I] is not valid input to mwrank (entries should be
↳integers)

```

A string of the form '[a1,a2,a3,a4,a6]' with any whitespace and integers ai:

```

sage: validate_mwrank_input('0 -1 1 -7 6')
'[0,-1,1,-7,6]'
sage: validate_mwrank_input("[0,-1,1,0,0]\n")
'[0,-1,1,0,0]'
sage: validate_mwrank_input('0\t -1\t 1\t 0\t 0\n')
'[0,-1,1,0,0]'
sage: validate_mwrank_input('0 -1 1 -7 ')
Traceback (most recent call last):
...
ValueError: 0 -1 1 -7 is not valid input to mwrank

```


INTERFACE TO GNU OCTAVE

GNU Octave is a free software (GPL) MATLAB-like program with numerical routines for integrating, solving systems of equations, special functions, and solving (numerically) differential equations. Please see <http://octave.org/> for more details.

The commands in this section only work if you have the optional “octave” interpreter installed and available in your PATH. It’s not necessary to install any special Sage packages.

EXAMPLES:

```
sage: octave.eval('2+2')    # optional - octave
'ans = 4'

sage: a = octave(10)      # optional - octave
sage: a**10               # optional - octave
1e+10
```

LOG: - creation (William Stein) - ? (David Joyner, 2005-12-18) - Examples (David Joyner, 2005-01-03)

33.1 Computation of Special Functions

Octave implements computation of the following special functions (see the maxima and gp interfaces for even more special functions):

```
airy
  Airy functions of the first and second kind, and their derivatives.
  airy(0,x) = Ai(x), airy(1,x) = Ai'(x), airy(2,x) = Bi(x), airy(3,x) = Bi'(x)
besselj
  Bessel functions of the first kind.
bessely
  Bessel functions of the second kind.
besseli
  Modified Bessel functions of the first kind.
besselk
  Modified Bessel functions of the second kind.
besselh
  Compute Hankel functions of the first (k = 1) or second (k = 2) kind.
beta
  The Beta function,
      beta(a, b) = gamma(a) * gamma(b) / gamma(a + b).
betainc
```

(continues on next page)

(continued from previous page)

```

    The incomplete Beta function,
erf
    The error function,
erfinv
    The inverse of the error function.
gamma
    The Gamma function,
gammainc
    The incomplete gamma function,

```

For example,

```

sage: octave("airy(3,2)")      # optional - octave
4.10068
sage: octave("beta(2,2)")     # optional - octave
0.166667
sage: octave("betainc(0.2,2,2)") # optional - octave
0.104
sage: octave("besselh(0,2)")  # optional - octave
(0.223891,0.510376)
sage: octave("besselh(0,1)")  # optional - octave
(0.765198,0.088257)
sage: octave("besseli(1,2)")  # optional - octave
1.59064
sage: octave("besselj(1,2)")  # optional - octave
0.576725
sage: octave("besselk(1,2)")  # optional - octave
0.139866
sage: octave("erf(0)")        # optional - octave
0
sage: octave("erf(1)")        # optional - octave
0.842701
sage: octave("erfinv(0.842)")  # optional - octave
0.998315
sage: octave("gamma(1.5)")    # optional - octave
0.886227
sage: octave("gammainc(1.5,1)") # optional - octave
0.77687

```

33.2 Tutorial

EXAMPLES:

```

sage: octave('4+10')          # optional - octave
14
sage: octave('date')          # optional - octave; random output
18-Oct-2007
sage: octave('5*10 + 6')     # optional - octave
56
sage: octave('(6+6)/3')      # optional - octave

```

(continues on next page)

(continued from previous page)

```
4
sage: octave('9')^2          # optional - octave
81
sage: a = octave(10); b = octave(20); c = octave(30)    # optional - octave
sage: avg = (a+b+c)/3      # optional - octave
sage: avg                  # optional - octave
20
sage: parent(avg)         # optional - octave
Octave
```

```
sage: my_scalar = octave('3.1415')    # optional - octave
sage: my_scalar                       # optional - octave
3.1415
sage: my_vector1 = octave('[1,5,7]')  # optional - octave
sage: my_vector1                     # optional - octave
1      5      7
sage: my_vector2 = octave('[1;5;7]')  # optional - octave
sage: my_vector2                     # optional - octave
1
5
7
sage: my_vector1 * my_vector2        # optional - octave
75
```

class sage.interfaces.octave.Octave(maxread=None, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None, seed=None, command=None)

Bases: *sage.interfaces.expect.Expect*

Interface to the Octave interpreter.

EXAMPLES:

```
sage: octave.eval("a = [ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]")    # optional - octave
'a =\n\n 1 1 2\n 3 5 8\n 13 21 33\n\n'
sage: octave.eval("b = [ 1; 3; 13]")                      # optional - octave
'b =\n\n 1\n 3\n 13\n\n'
sage: octave.eval("c=a \\ b") # solves linear equation: a*c = b # optional -
↪octave; random output
'c =\n\n 1\n 7.21645e-16\n -7.21645e-16\n\n'
sage: octave.eval("c")                                    # optional - octave; random
↪output
'c =\n\n 1\n 7.21645e-16\n -7.21645e-16\n\n'
```

clear(var)

Clear the variable named var.

EXAMPLES:

```
sage: octave.set('x', '2') # optional - octave
sage: octave.clear('x')   # optional - octave
sage: octave.get('x')     # optional - octave
"error: 'x' undefined near line ... column 1"
```

console()

Spawn a new Octave command-line session.

This requires that the optional octave program be installed and in your PATH, but no optional Sage packages need be installed.

EXAMPLES:

```
sage: octave_console()          # not tested
GNU Octave, version 2.1.73 (i386-apple-darwin8.5.3).
Copyright (C) 2006 John W. Eaton.
...
octave:1> 2+3
ans = 5
octave:2> [ctl-d]
```

Pressing ctrl-d exits the octave console and returns you to Sage. octave, like Sage, remembers its history from one session to another.

de_system_plot(f, ics, trange)

Plot (using octave's interface to gnuplot) the solution to a 2×2 system of differential equations.

INPUT:

- f - a pair of strings representing the differential equations; The independent variable must be called x and the dependent variable must be called y.
- ics - a pair [x0,y0] such that $x(t_0) = x_0, y(t_0) = y_0$
- trange - a pair [t0,t1]

OUTPUT: a gnuplot window appears

EXAMPLES:

```
sage: octave.de_system_plot(['x+y', 'x-y'], [1,-1], [0,2]) # not tested -- does_
↪this actually work (on OS X it fails for me -- William Stein, 2007-10)
```

This should yield the two plots $(t, x(t)), (t, y(t))$ on the same graph (the t-axis is the horizontal axis) of the system of ODEs

$$x' = x + y, x(0) = 1; \quad y' = x - y, y(0) = -1, \quad \text{for } 0 < t < 2.$$

get(var)

Get the value of the variable var.

EXAMPLES:

```
sage: octave.set('x', '2') # optional - octave
sage: octave.get('x') # optional - octave
' 2'
```

quit(verbose=False)

EXAMPLES:

```
sage: o = Octave()
sage: o._start() # optional - octave
sage: o.quit(True) # optional - octave
Exiting spawned Octave process.
```

sage2octave_matrix_string(A)

Return an octave matrix from a Sage matrix.

INPUT: A Sage matrix with entries in the rationals or reals.

OUTPUT: A string that evaluates to an Octave matrix.

EXAMPLES:

```
sage: M33 = MatrixSpace(QQ,3,3)
sage: A = M33([1,2,3,4,5,6,7,8,0])
sage: octave.sage2octave_matrix_string(A) # optional - octave
'[1, 2, 3; 4, 5, 6; 7, 8, 0]'
```

AUTHORS:

- David Joyner and William Stein

set(var, value)

Set the variable `var` to the given value.

EXAMPLES:

```
sage: octave.set('x', '2') # optional - octave
sage: octave.get('x') # optional - octave
' 2'
```

set_seed(seed=None)

Set the seed for the random number generator for this octave interpreter.

EXAMPLES:

```
sage: o = Octave() # optional - octave
sage: o.set_seed(1) # optional - octave
1
sage: [o.rand() for i in range(5)] # optional - octave
[ 0.134364,  0.847434,  0.763775,  0.255069,  0.495435]
```

solve_linear_system(A, b)

Use octave to compute a solution x to $A*x = b$, as a list.

INPUT:

- A – $m \times n$ matrix A with entries in \mathbf{Q} or \mathbf{R}
- b – m -vector b entries in \mathbf{Q} or \mathbf{R} (resp)

OUTPUT: A list x (if it exists) which solves $M*x = b$

EXAMPLES:

```
sage: M33 = MatrixSpace(QQ,3,3)
sage: A = M33([1,2,3,4,5,6,7,8,0])
sage: V3 = VectorSpace(QQ,3)
sage: b = V3([1,2,3])
sage: octave.solve_linear_system(A,b) # optional - octave (and output is
↳slightly random in low order bits)
[-0.33333299999999999, 0.6666670000000000001, -3.52366000000000002e-18]
```

AUTHORS:

- David Joyner and William Stein

version()

Return the version of Octave.

OUTPUT: string

EXAMPLES:

```
sage: v = octave.version() # optional - octave
sage: v                     # optional - octave; random
'2.13.7'

sage: import re
sage: assert re.match(r"\d+\.\d+\.\d+", v) is not None # optional - octave
```

class `sage.interfaces.octave.OctaveElement`(*parent, value, is_name=False, name=None*)

Bases: `sage.interfaces.expect.ExpectElement`

`sage.interfaces.octave.octave_console()`

Spawn a new Octave command-line session.

This requires that the optional octave program be installed and in your PATH, but no optional Sage packages need be installed.

EXAMPLES:

```
sage: octave_console() # not tested
GNU Octave, version 2.1.73 (i386-apple-darwin8.5.3).
Copyright (C) 2006 John W. Eaton.
...
octave:1> 2+3
ans = 5
octave:2> [ctl-d]
```

Pressing ctrl-d exits the octave console and returns you to Sage. octave, like Sage, remembers its history from one session to another.

`sage.interfaces.octave.reduce_load_Octave()`

EXAMPLES:

```
sage: from sage.interfaces.octave import reduce_load_Octave
sage: reduce_load_Octave()
Octave
```

`sage.interfaces.octave.to_complex`(*octave_string, R*)

Helper function to convert octave complex number

INTERFACE TO PHC.

PHC computes numerical information about systems of polynomials over the complex numbers.

PHC implements polynomial homotopy methods to exploit structure in order to better approximate all isolated solutions. The package also includes extra tools to handle positive dimensional solution components.

AUTHORS:

- PHC was written by J. Verschelde, R. Cools, and many others (?)
- William Stein and Kelly ?? – first version of interface to PHC
- Marshall Hampton – second version of interface to PHC
- Marshall Hampton and Alex Jokela – third version, path tracking

class `sage.interfaces.phc.PHC`

Bases: `object`

A class to interface with PHCpack, for computing numerical homotopies and root counts.

EXAMPLES:

```

sage: from sage.interfaces.phc import phc
sage: R.<x,y> = PolynomialRing(CDF,2)
sage: testsys = [x^2 + 1, x*y - 1]
sage: phc.mixed_volume(testsys)          # optional -- phc
2
sage: v = phc.blackbox(testsys, R)      # optional -- phc
sage: sols = v.solutions()             # optional -- phc
sage: sols.sort()                      # optional -- phc
sage: sols                             # optional -- phc
[[-1.0000000000000000*I, 1.0000000000000000*I], [1.0000000000000000*I, -1.
↪0000000000000000*I]]
sage: sol_dict = v.solution_dicts()    # optional -- phc
sage: x_sols_from_dict = [d[x] for d in sol_dict] # optional -- phc
sage: x_sols_from_dict.sort(); x_sols_from_dict # optional -- phc
[-1.0000000000000000*I, 1.0000000000000000*I]
sage: residuals = [[test_equation.change_ring(CDF).subs(sol) for test_equation in_
↪testsys] for sol in v.solution_dicts()] # optional -- phc
sage: residuals                        # optional -- phc
[[0, 0], [0, 0]]

```

blackbox(*polys, input_ring, verbose=False*)

Return as a string the result of running PHC with the given polynomials under blackbox mode (the ‘-b’ option).

INPUT:

- `polys` – a list of multivariate polynomials (elements of a multivariate polynomial ring).
- `input_ring` – for coercion of the variables into the desired ring.
- `verbose` – print lots of verbose information about what this function does.

OUTPUT:

- a `PHC_Object` object containing the `phcpack` output string.

EXAMPLES:

```
sage: from sage.interfaces.phc import *
sage: R2.<x,y> = PolynomialRing(QQ,2)
sage: start_sys = [x^6-y^2,y^5-1]
sage: sol = phc.blackbox(start_sys, R2)           # optional -- phc
sage: len(sol.solutions())                       # optional -- phc
30
```

`mixed_volume`(*polys*, *verbose=False*)

Compute the mixed volume of the polynomial system given by the input `polys`.

INPUT:

- `polys` – a list of multivariate polynomials (elements of a multivariate polynomial ring).
- `verbose` – print lots of verbose information about what this function does.

OUTPUT:

- The mixed volume.

EXAMPLES:

```
sage: from sage.interfaces.phc import *
sage: R2.<x,y,z> = PolynomialRing(QQ,3)
sage: test_sys = [(x+y+z)^2-1,x^2-x,y^2-1]
sage: phc.mixed_volume(test_sys)                 # optional -- phc
4
```

`path_track`(*start_sys*, *end_sys*, *input_ring*, *c_skew=0.001*, *saved_start=None*)

This function computes homotopy paths between the solutions of `start_sys` and `end_sys`.

INPUT:

- `start_sys` – a square polynomial system, given as a list of polynomials
- `end_sys` – same type as `start_sys`
- `input_ring` – for coercion of the variables into the desired ring.
- `c_skew` – optional. the imaginary part of homotopy multiplier; nonzero values are often necessary to avoid intermediate path collisions
- `saved_start` – optional. A `phc` output file. If not given, start system solutions are computed via the `phc.blackbox` function.

OUTPUT:

- a list of paths as dictionaries, with the keys `variables` and `t-values` on the path.

EXAMPLES:

```

sage: from sage.interfaces.phc import *
sage: R2.<x,y> = PolynomialRing(QQ,2)
sage: start_sys = [x^6-y^2,y^5-1]
sage: sol = phc.blackbox(start_sys, R2)           # optional -- phc
sage: start_save = sol.save_as_start()           # optional -- phc
sage: end_sys = [x^7-2,y^5-x^2]                 # optional -- phc
sage: sol_paths = phc.path_track(start_sys, end_sys, R2, saved_start = start_
↪save) # optional -- phc
sage: len(sol_paths)                            # optional -- phc
30
    
```

plot_paths_2d(*start_sys, end_sys, input_ring, c_skew=0.001, endpoints=True, saved_start=None, rand_colors=False*)

Return a graphics object of solution paths in the complex plane.

INPUT:

- *start_sys* – a square polynomial system, given as a list of polynomials
- *end_sys* – same type as *start_sys*
- *input_ring* – for coercion of the variables into the desired ring.
- *c_skew* – optional. the imaginary part of homotopy multiplier; nonzero values are often necessary to avoid intermediate path collisions
- *endpoints* – optional. Whether to draw in the ends of paths as points.
- *saved_start* – optional. A phc output file. If not given, start system solutions are computed via the `phc.blackbox` function.

OUTPUT:

- lines and points of solution paths

EXAMPLES:

```

sage: from sage.interfaces.phc import *
sage: from sage.structure.sage_object import SageObject
sage: R2.<x,y> = PolynomialRing(QQ,2)
sage: start_sys = [x^5-y^2,y^5-1]
sage: sol = phc.blackbox(start_sys, R2)         # optional -- phc
sage: start_save = sol.save_as_start()         # optional -- phc
sage: end_sys = [x^5-25,y^5-x^2]               # optional -- phc
sage: testing = phc.plot_paths_2d(start_sys, end_sys, R2) # optional -- phc
sage: type(testing)                            # optional -- phc (normally use plot_
↪here)
<class 'sage.plot.graphics.Graphics'>
    
```

start_from(*start_filename_or_string, polys, input_ring, path_track_file=None, verbose=False*)

This computes solutions starting from a phcpack solution file.

INPUT:

- *start_filename_or_string* – the filename for a phcpack start system, or the contents of such a file as a string. Variable names must match the inputring variables. The value of the homotopy variable *t* should be 1, not 0.
- *polys* – a list of multivariate polynomials (elements of a multivariate polynomial ring).
- *input_ring*: for coercion of the variables into the desired ring.

- path_track_file: whether to save path-tracking information
- verbose – print lots of verbose information about what this function does.

OUTPUT:

- A solution in the form of a PHCObject.

EXAMPLES:

```
sage: from sage.interfaces.phc import *
sage: R2.<x,y> = PolynomialRing(QQ,2)
sage: start_sys = [x^6-y^2,y^5-1]
sage: sol = phc.blackbox(start_sys, R2) # optional -- phc
sage: start_save = sol.save_as_start() # optional -- phc
sage: end_sys = [x^7-2,y^5-x^2] # optional -- phc
sage: sol = phc.start_from(start_save, end_sys, R2) # optional -- phc
sage: len(sol.solutions()) # optional -- phc
30
```

class sage.interfaces.phc.PHC_Object(*output_file_contents, input_ring*)

Bases: object

A container for data from the PHCpack program - lists of float solutions, etc. Currently the file contents are kept as a string; for really large outputs this would be bad.

INPUT:

- output_file_contents: the string output of PHCpack
- input_ring: for coercion of the variables into the desired ring.

EXAMPLES:

```
sage: from sage.interfaces.phc import phc
sage: R2.<x,y> = PolynomialRing(QQ,2)
sage: start_sys = [(x-1)^2+(y-1)-1, x^2+y^2-1]
sage: sol = phc.blackbox(start_sys, R2) # optional -- phc
sage: str(sum([x[0] for x in sol.solutions()]).real())[0:3] # optional -- phc
'2.0'
```

classified_solution_dicts()

Return a dictionary of lists of dictionaries of solutions.

Its not as crazy as it sounds; the keys are the types of solutions as classified by phcpack: regular vs. singular, complex vs. real

INPUT:

- None

OUTPUT:

- A dictionary of lists of dictionaries of solutions

EXAMPLES:

```
sage: from sage.interfaces.phc import phc
sage: R.<x,y> = PolynomialRing(CC,2)
sage: p_sys = [x^10-y,y^2-1]
sage: sol = phc.blackbox(p_sys,R) # optional -- phc
```

(continues on next page)

(continued from previous page)

```
sage: classifieds = sol.classified_solution_dicts() # optional -- phc
sage: str(sum([q[y] for q in classifieds['real']]))[0:3] # optional -- phc
'2.0'
```

save_as_start(start_filename=None, sol_filter="")

Save a solution as a phcpack start file. The usual output is just as a string, but it can be saved to a file as well. Even if saved to a file, it still returns the output string.

EXAMPLES:

```
sage: from sage.interfaces.phc import phc
sage: R2.<x,y> = PolynomialRing(QQ,2)
sage: start_sys = [x^3-y^2,y^5-1]
sage: sol = phc.blackbox(start_sys, R2) # optional -- phc
sage: start_save = sol.save_as_start() # optional -- phc
sage: end_sys = [x^7-2,y^5-x^2] # optional -- phc
sage: sol = phc.start_from(start_save, end_sys, R2) # optional -- phc
sage: len(sol.solutions()) # optional -- phc
15
```

solution_dicts(get_failures=False)

Return a list of solutions in dictionary form: variable:value.

INPUT:

- self – for access to self_out_file_contents, the string of raw PHCpack output.
- get_failures (optional) – a boolean. The default (False) is to not process failed homotopies. These either lie on positive-dimensional components or at infinity.

OUTPUT:

- solution_dicts: a list of dictionaries. Each dictionary element is of the form variable:value, where the variable is an element of the input_ring, and the value is in ComplexField.

EXAMPLES:

```
sage: from sage.interfaces.phc import *
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: fs = [x^2-1,y^2-x,z^2-y]
sage: sol = phc.blackbox(fs,R) # optional -- phc
sage: s_list = sol.solution_dicts() # optional -- phc
sage: s_list.sort() # optional -- phc
sage: s_list[0] # optional -- phc
{y: 1.0000000000000000, z: -1.0000000000000000, x: 1.0000000000000000}
```

solutions(get_failures=False)

Return a list of solutions in the ComplexField.

Use the variable_list function to get the order of variables used by PHCpack, which is usually different than the term order of the input_ring.

INPUT:

- self – for access to self_out_file_contents, the string of raw PHCpack output.
- get_failures (optional) – a boolean. The default (False) is to not process failed homotopies. These either lie on positive-dimensional components or at infinity.

OUTPUT:

- solutions: a list of lists of ComplexField-valued solutions.

EXAMPLES:

```
sage: from sage.interfaces.phc import *
sage: R2.<x1,x2> = PolynomialRing(QQ,2)
sage: test_sys = [x1^5-x1*x2^2-1, x2^5-x1*x2-1]
sage: sol = phc.blackbox(test_sys, R2)           # optional -- phc
sage: len(sol.solutions())                       # optional -- phc
25
```

variable_list()

Return the variables, as strings, in the order in which PHCpack has processed them.

EXAMPLES:

```
sage: from sage.interfaces.phc import *
sage: R2.<x1,x2> = PolynomialRing(QQ,2)
sage: test_sys = [x1^5-x1*x2^2-1, x2^5-x1*x2-1]
sage: sol = phc.blackbox(test_sys, R2)           # optional -- phc
sage: sol.variable_list()                       # optional -- phc
['x1', 'x2']
```

sage.interfaces.phc.get_classified_solution_dicts(*output_file_contents*, *input_ring*, *get_failures=True*)

Return a dictionary of lists of dictionaries of variable:value (key:value) pairs. Only used internally; see the `classified_solution_dict` function in the `PHC_Object` class definition for details.

INPUT:

- *output_file_contents* – phc solution output as a string
- *input_ring* – a PolynomialRing that variable names can be coerced into

OUTPUT:

- a dictionary of lists if dictionaries of solutions, classifies by type

EXAMPLES:

```
sage: from sage.interfaces.phc import *
sage: R2.<x1,x2> = PolynomialRing(QQ,2)
sage: test_sys = [(x1-2)^5-x2, (x2-1)^5-1]
sage: sol = phc.blackbox(test_sys, R2)           # optional -- phc
sage: sol_classes = get_classified_solution_dicts(sol.output_file_contents,R2) #_
↳optional -- phc
sage: len(sol_classes['real'])                   # optional -- phc
1
```

sage.interfaces.phc.get_solution_dicts(*output_file_contents*, *input_ring*, *get_failures=True*)

Return a list of dictionaries of variable:value (key:value) pairs. Only used internally; see the `solution_dict` function in the `PHC_Object` class definition for details.

INPUT:

- *output_file_contents* – phc solution output as a string
- *input_ring* – a PolynomialRing that variable names can be coerced into

OUTPUT:

a list of dictionaries of solutions

EXAMPLES:

```
sage: from sage.interfaces.phc import *
sage: R2.<x1,x2> = PolynomialRing(QQ,2)
sage: test_sys = [(x1-1)^5-x2, (x2-1)^5-1]
sage: sol = phc.blackbox(test_sys, R2) # optional -- phc
sage: test = get_solution_dicts(sol.output_file_contents,R2) # optional -- phc
sage: str(sum([q[x1].real() for q in test]))[0:4] # optional -- phc
'25.0'
```

`sage.interfaces.phc.get_variable_list(output_file_contents)`

Return the variables, as strings, in the order in which PHCpack has processed them.

EXAMPLES:

```
sage: from sage.interfaces.phc import *
sage: R2.<x1,x2> = PolynomialRing(QQ,2)
sage: test_sys = [(x1-2)^5-x2, (x2-1)^5-1]
sage: sol = phc.blackbox(test_sys, R2) # optional -- phc
sage: get_variable_list(sol.output_file_contents) # optional -- phc
['x1', 'x2']
```

`sage.interfaces.phc.random(*args, **kws)`

`random()` -> x in the interval [0, 1).

INTERFACE TO POLYMAKE

polymake (<https://polymake.org>) is a mature open source package for research in polyhedral geometry and related fields, developed since 1997 by Evgenij Gawrilow and Michael Joswig and various contributors.

polymake has been described in [GJ1997], [GJ2006], [JMP2009], [GJRW2010], [GHJ2016], and [AGHJLPR2017].

`sage.interfaces.polymake.Polymake`
alias of `sage.interfaces.polymake.PolymakeJuPyMake`

class `sage.interfaces.polymake.PolymakeAbstract`(*seed=None*)
Bases: `sage.interfaces.tab_completion.ExtraTabCompletion`, `sage.interfaces.interface.Interface`

Abstract interface to the polymake interpreter.

This class should not be instantiated directly, but through its subclasses `Polymake` (Pexpect interface) or `PolymakeJuPyMake` (JuPyMake interface).

EXAMPLES:

```
sage: from sage.interfaces.polymake import PolymakeAbstract, polymake_jupymake
```

We test the verbosity management with very early doctests because messages will not be repeated.

Testing the JuPyMake interface:

```
sage: isinstance(polymake_jupymake, PolymakeAbstract)
True
sage: p = polymake_jupymake.rand_sphere(4, 20, seed=5)      # optional - jupymake
sage: p                                                    # optional - jupymake
Random spherical polytope of dimension 4; seed=5...
sage: set_verbosity(3)
sage: p.H_VECTOR                                          # optional - jupymake
polymake: used package ppl
The Parma Polyhedra Library ...
1 16 40 16 1
sage: set_verbosity(0)
sage: p.F_VECTOR                                          # optional - jupymake
20 94 148 74
```

application(*app*)

Change to a given polymake application.

INPUT:

- *app*, a string, one of “common”, “fulton”, “group”, “matroid”, “topaz”, “fan”, “graph”, “ideal”, “polytope”, “tropical”

EXAMPLES:

We expose a computation that uses both the ‘polytope’ and the ‘fan’ application of polymake. Let us start by defining a polytope q in terms of inequalities. Polymake knows to compute the f- and h-vector and finds that the polytope is very ample:

```
sage: q = polymake.new_object("Polytope", INEQUALITIES=[[5,-4,0,1],[-3,0,-4,1],
↳ [-2,1,0,0],[-4,4,4,-1],[0,0,1,0],[8,0,0,-1],[1,0,-1,0],[3,-1,0,0]]) #_
↳ optional - jupymake
sage: q.H_VECTOR # optional - jupymake
1 5 5 1
sage: q.F_VECTOR # optional - jupymake
8 14 8
sage: q.VERY_AMPLE # optional - jupymake
true
```

In the application ‘fan’, polymake can now compute the normal fan of q and its (primitive) rays:

```
sage: polymake.application('fan') # optional - jupymake
sage: g = q.normal_fan() # optional - jupymake
sage: g.RAYS # optional - jupymake
-1 0 1/4
0 -1 1/4
1 0 0
1 1 -1/4
0 1 0
0 0 -1
0 -1 0
-1 0 0
sage: g.RAYS.primitive() # optional - jupymake
-4 0 1
0 -4 1
1 0 0
4 4 -1
0 1 0
0 0 -1
0 -1 0
-1 0 0
```

Note that the list of functions available by tab completion depends on the application.

clear(*var*)

Clear the variable named *var*.

Note: This is implicitly done when deleting an element in the interface.

console()

Raise an error, pointing to *interact()* and *polymake_console()*.

EXAMPLES:

```
sage: polymake.console()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
NotImplementedError: Please use polymake_console() function or the .interact()
↳method
```

function_call(function, args=None, kwds=None)

EXAMPLES:

```
sage: polymake.rand_sphere(4, 30, seed=15) # optional - jupymake #
↳indirect doctest
Random spherical polytope of dimension 4; seed=15...
```

get(cmd)

Return the string representation of an object in the polymake interface.

EXAMPLES:

```
sage: polymake.get('cube(3)') # optional - jupymake
'Polymake::polytope::Polytope__Rational=ARRAY(...)'
```

Note that the above string representation is what polymake provides. In our interface, we use what polymake calls a “description”:

```
sage: polymake('cube(3)') # optional - jupymake
cube of dimension 3
```

help(topic, pager=True)

Displays polymake’s help on a given topic, as a string.

INPUT:

- topic, a string
- pager, optional bool, default True: When True, display help, otherwise return as a string.

EXAMPLES:

```
sage: print(polymake.help('Polytope', pager=False)) # optional -
↳jupymake # random
objects/Polytope:
Not necessarily bounded or unbounded polyhedron.
Nonetheless, the name "Polytope" is used for two reasons:
Firstly, combinatorially we always deal with polytopes; see the description of
↳VERTICES_IN_FACETS for details.
The second reason is historical.
We use homogeneous coordinates, which is why Polytope is derived from Cone.
Note that a pointed polyhedron is projectively equivalent to a polytope.
Scalar is the numeric data type used for the coordinates.
```

In some cases, polymake expects user interaction to choose from different available help topics. In these cases, a warning is given, and the available help topics are displayed resp. printed, without user interaction:

```
sage: polymake.help('TRIANGULATION') # optional -
↳jupymake # random
doctest:warning
...
```

(continues on next page)

(continued from previous page)

```
UserWarning: Polymake expects user interaction. We abort and return the options_
↳ that Polymake provides.
There are 5 help topics matching 'TRIANGULATION':
1: objects/Visualization/Visual::Polytope/methods/TRIANGULATION
2: objects/Visualization/Visual::PointConfiguration/methods/TRIANGULATION
3: objects/Cone/properties/Triangulation and volume/TRIANGULATION
4: objects/PointConfiguration/properties/Triangulation and volume/TRIANGULATION
5: objects/Polytope/properties/Triangulation and volume/TRIANGULATION
```

If an unknown help topic is requested, a *PolymakeError* results:

```
sage: polymake.help('Triangulation')      # optional - jupymake
Traceback (most recent call last):
...
PolymakeError: unknown help topic 'Triangulation'
```

new_object(*name*, **args*, ***kws*)

Return a new instance of a given polymake type, with given positional or named arguments.

INPUT:

- name of a polymake class (potentially templated), as string.
- further positional or named arguments, to be passed to the constructor.

EXAMPLES:

```
sage: q = polymake.new_object("Polytope<Rational>", INEQUALITIES=[[4,-4,0,1],[
↳ -4,0,-4,1],[-2,1,0,0],[-4,4,4,-1],[0,0,1,0],[8,0,0,-1]]) # optional - jupymake
sage: q.N_VERTICES      # optional - jupymake
4
sage: q.BOUNDED        # optional - jupymake
true
sage: q.VERTICES      # optional - jupymake
1 2 0 4
1 3 0 8
1 2 1 8
1 3 1 8
sage: q.full_typename() # optional - jupymake
'Polytope<Rational>'
```

set(*var*, *value*)

Set the variable *var* to the given value.

Eventually, *var* is a reference to *value*.

Warning: This method, although it doesn't start with an underscore, is an internal method and not part of the interface. So, please do not try to call it explicitly. Instead, use the polymake interface as shown in the examples.

REMARK:

Polymake's user language is Perl. In Perl, if one wants to assign the return value of a function to a variable, the syntax to do so depends on the type of the return value. While this is fine in compiled code, it seems quite awkward in user interaction.

To make this polymake pexpect interface a bit more user friendly, we treat *all* variables as arrays. A scalar value (most typically a reference) is thus interpreted as the only item in an array of length one. It is, of course, possible to use the interface without knowing these details.

EXAMPLES:

```
sage: c = polymake('cube(3)') # optional - jupymake #_
↪ indirect doctest
sage: d = polymake.cube(3) # optional - jupymake
```

Equality is, for “big” objects such as polytopes, comparison by identity:

```
sage: c == d # optional - jupymake
False
```

However, the list of vertices is equal:

```
sage: c.VERTICALICES == d.VERTICALICES # optional - jupymake
True
```

version()

Version of the polymake installation.

EXAMPLES:

```
sage: polymake.version() # optional - jupymake # random
'4...'
```

class sage.interfaces.polymake.PolymakeElement(parent, value, is_name=False, name=None)

Bases: `sage.interfaces.tab_completion.ExtraTabCompletion`, `sage.interfaces.interface.InterfaceElement`

Elements in the polymake interface.

EXAMPLES:

We support all “big” polymake types, Perl arrays of length different from one, and Perl scalars:

```
sage: p = polymake.rand_sphere(4, 20, seed=5) # optional - jupymake
sage: p.typename() # optional - jupymake
'Polytope'
sage: p # optional - jupymake
Random spherical polytope of dimension 4; seed=5...
```

Now, one can work with that element in Python syntax, for example:

```
sage: p.VERTICALICES[2][2] # optional - jupymake
1450479926727001/2251799813685248
```

full_typename()

The name of the specialised type of this element.

EXAMPLES:

```
sage: c = polymake.cube(4) # optional - jupymake
sage: c.full_typename() # optional - jupymake
'Polytope<Rational>'
```

(continues on next page)

(continued from previous page)

```
sage: c.VERTICALS.full_typename() # optional - jupymake
'Matrix<Rational, NonSymmetric>'
```

get_member(*attrname*)

Get a member/property of this element.

Note: Normally, it should be possible to just access the property in the usual Python syntax for attribute access. However, if that fails, one can request the member explicitly.

EXAMPLES:

```
sage: p = polymake.rand_sphere(4, 20, seed=5) # optional - jupymake
```

Normally, a property would be accessed as follows:

```
sage: p.F_VECTOR # optional - jupymake
20 94 148 74
```

However, explicit access is possible as well:

```
sage: p.get_member('F_VECTOR') # optional - jupymake
20 94 148 74
```

In some cases, the explicit access works better:

```
sage: p.type # optional - jupymake
Member function 'type' of Polymake::polytope::Polytope__Rational object
sage: p.get_member('type') # optional - jupymake
Polytope<Rational>[SAGE...]
sage: p.get_member('type').get_member('name') # optional - jupymake
Polytope
```

Note that in the last example calling the erroneously constructed member function `type` still works:

```
sage: p.type() # optional - jupymake
Polytope<Rational>[SAGE...]
```

get_member_function(*attrname*)

Request a member function of this element.

Note: It is not checked whether a member function with the given name exists.

EXAMPLES:

```
sage: c = polymake.cube(2) # optional - jupymake
↪ jupymake
sage: c.contains # optional - jupymake
↪ jupymake
Member function 'contains' of Polymake::polytope::Polytope__Rational object
sage: V = polymake.new_object('Vector', [1,0,0]) # optional - jupymake
↪ jupymake
```

(continues on next page)

(continued from previous page)

```
sage: V # optional -j
↪ jupymake
1 0 0
sage: c.contains(V) # optional -j
↪ jupymake
true
```

Whether a member function of the given name actually exists for that object will only be clear when calling it:

```
sage: c.get_member_function("foo") # optional -j
↪ jupymake
Member function 'foo' of Polymake::polytope::Polytope__Rational object
sage: c.get_member_function("foo")() # optional -j
↪ jupymake
Traceback (most recent call last):
...
TypeError: Can't locate object method "foo" via package
↪ "Polymake::polytope::Polytope__Rational"
```

known_properties()

List the names of properties that have been computed so far on this element.

Note: This is in many cases equivalent to use polymake’s `list_properties`, which returns a blank separated string representation of the list of properties. However, on some elements, `list_properties` would simply result in an error.

EXAMPLES:

```
sage: c = polymake.cube(4) # optional - jupymake
sage: c.known_properties() # optional - jupymake
['AFFINE_HULL',
 'BOUNDED',
 'CONE_AMBIENT_DIM',
 'CONE_DIM',
 ...
 'VERTICES_IN_FACETS']
sage: c.list_properties() # optional - jupymake
CONE_AMBIENT_DIM, CONE_DIM, FACETS, AFFINE_HULL, VERTICES_IN_FACETS,
BOUNDED...
```

A computation can change the list of known properties:

```
sage: c.F_VECTOR # optional - jupymake
16 32 24 8
sage: c.known_properties() # optional - jupymake
['AFFINE_HULL',
 'BOUNDED',
 'COMBINATORIAL_DIM',
 'CONE_AMBIENT_DIM',
 'CONE_DIM',
```

(continues on next page)

(continued from previous page)

```
...
'VERTICES_IN_FACETS']
```

qualified_typename()

The qualified name of the type of this element.

EXAMPLES:

```
sage: c = polymake.cube(4) # optional - jupymake
sage: c.qualified_typename() # optional - jupymake
'polytope::Polytope<Rational>'
sage: c.VERTICALES.qualified_typename() # optional - jupymake
'common::Matrix<Rational, NonSymmetric>'
```

typename()

The name of the underlying base type of this element in polymake.

EXAMPLES:

```
sage: c = polymake.cube(4) # optional - jupymake
sage: c.typename() # optional - jupymake
'Polytope'
sage: c.VERTICALES.typename() # optional - jupymake
'Matrix'
```

typeof()

Return the type of a polymake “big” object, and its underlying Perl type.

Note: This is mainly for internal use.

EXAMPLES:

```
sage: p = polymake.rand_sphere(3, 13, seed=12) # optional - ↵
↵ jupymake
sage: p.typeof() # optional - ↵
↵ jupymake
('Polymake::polytope::Polytope__Rational', 'ARRAY')
sage: p.VERTICALES.typeof() # optional - ↵
↵ jupymake
('Polymake::common::Matrix_A_Rational_I_NonSymmetric_Z', 'ARRAY')
sage: p.get_schedule('F_VECTOR').typeof() # optional - ↵
↵ jupymake
('Polymake::Core::Scheduler::RuleChain', 'ARRAY')
```

On “small” objects, it just returns empty strings:

```
sage: p.N_VERTICALES.typeof() # optional - ↵
↵ jupymake
('', '')
sage: p.list_properties().typeof() # optional - ↵
↵ jupymake
('', '')
```

exception `sage.interfaces.polymake.PolymakeError`

Bases: `RuntimeError`

Raised if polymake yields an error message.

class `sage.interfaces.polymake.PolymakeFunctionElement`(*obj, name, memberfunction=False*)

Bases: `sage.interfaces.interface.InterfaceFunctionElement`

A callable (function or member function) bound to a polymake element.

EXAMPLES:

```
sage: c = polymake.cube(2) # optional - jupymake
sage: V = polymake.new_object('Vector', [1,0,0]) # optional - jupymake
sage: V # optional - jupymake
1 0 0
sage: c.contains # optional - jupymake
Member function 'contains' of Polymake::polytope::Polytope__Rational object
sage: c.contains(V) # optional - jupymake
true
```

class `sage.interfaces.polymake.PolymakeJuPyMake`(*seed=None, verbose=False*)

Bases: `sage.interfaces.polymake.PolymakeAbstract`

Interface to the polymake interpreter using JuPyMake.

In order to use this interface, you need to either install the optional polymake package for Sage, or install polymake system-wide on your computer; it is available from <https://polymake.org>. Also install the jupymake Python package.

Type `polymake.[tab]` for a list of most functions available from your polymake install. Type `polymake.Function?` for polymake's help about a given Function. Type `polymake(...)` to create a new polymake object, and `polymake.eval(...)` to run a string using polymake and get the result back as a string.

EXAMPLES:

```
sage: from sage.interfaces.polymake import polymake_jupymake as polymake
sage: type(polymake)
<...sage.interfaces.polymake.PolymakeJuPyMake...
sage: p = polymake.rand_sphere(4, 20, seed=5) # optional - jupymake
sage: p # optional - jupymake
Random spherical polytope of dimension 4; seed=5...
sage: set_verbose(3)
sage: p.H_VECTOR; # optional - jupymake # random
used package ppl
The Parma Polyhedra Library ...
sage: p.H_VECTOR # optional - jupymake
1 16 40 16 1
sage: set_verbose(0)
sage: p.F_VECTOR # optional - jupymake
20 94 148 74
sage: print(p.F_VECTOR._sage_doc_()) # optional - jupymake # random
property_types/Algebraic Types/Vector:
A type for vectors with entries of type Element.

You can perform algebraic operations such as addition or scalar multiplication.

You can create a new Vector by entering its elements, e.g.:
```

(continues on next page)

(continued from previous page)

```
$v = new Vector<Int>(1,2,3);
or
$v = new Vector<Int>([1,2,3]);
```

Python strings are translated to polymake (Perl) identifiers. To obtain Perl strings, use strings containing double-quote characters. Python dicts are translated to Perl hashes.

```
sage: L = polymake.db_query({'_id': 'F.4D.0047'}, # long time, optional -
↳jupymake internet perl_mongodb
.....: db="LatticePolytopes",
.....: collection="SmoothReflexive"); L
BigObjectArray
sage: len(L) # long time, optional -
↳jupymake internet perl_mongodb
1
sage: P = L[0] # long time, optional -
↳jupymake internet perl_mongodb
sage: sorted(P.list_properties(), key=str) # long time, optional -
↳jupymake internet perl_mongodb
[... , LATTICE_POINTS_GENERATORS, ... , POINTED, ...]
sage: P.F_VECTOR # long time, optional -
↳jupymake internet perl_mongodb
20 40 29 9
```

eval(code, **kws)

Evaluate a command.

INPUT:

- code – a command (string) to be evaluated

Different reaction types of polymake, including warnings, comments, errors, request for user interaction, and yielding a continuation prompt, are taken into account.

EXAMPLES:

```
sage: from sage.interfaces.polymake import polymake_jupymake as polymake
sage: p = polymake.cube(3) # optional - jupymake # indirect_
↳doctest
```

Here we see that remarks printed by polymake are displayed if the verbosity is positive:

```
sage: set_verbosity(1)
sage: p.N_LATTICE_POINTS # optional - jupymake # random
used package latte
LattE (Lattice point Enumeration) is a computer software dedicated to the
problems of counting lattice points and integration inside convex polytopes.
Copyright by Matthias Koeppel, Jesus A. De Loera and others.
http://www.math.ucdavis.edu/~latte/
27
sage: set_verbosity(0)
```

If polymake raises an error, the polymake *interface* raises a *PolymakeError*:

```

sage: polymake.eval('FOOBAR(3);')          # optional - jupymake
Traceback (most recent call last):
...
PolymakeError: Undefined subroutine &Polymake::User::FOOBAR called...
    
```

If a command is incomplete, then polymake returns a continuation prompt. In that case, we raise an error:

```

sage: polymake.eval('print 3')           # optional - jupymake
Traceback (most recent call last):
...
SyntaxError: Incomplete polymake command 'print 3'
sage: polymake.eval('print 3;')         # optional - jupymake
'3'
    
```

However, if the command contains line breaks but eventually is complete, no error is raised:

```

sage: print(polymake.eval('$tmp="abc";\nprint $tmp;')) # optional - jupymake
abc
    
```

When requesting help, polymake sometimes expect the user to choose from a list. In that situation, we abort with a warning, and show the list from which the user can choose; we could demonstrate this using the `help()` method, but here we use an explicit code evaluation:

```

sage: print(polymake.eval('help "TRIANGULATION";')) # optional - jupymake #
↳random
doctest:warning
...
UserWarning: Polymake expects user interaction. We abort and return
the options that Polymake provides.
There are 5 help topics matching 'TRIANGULATION':
1: objects/Cone/properties/Triangulation and volume/TRIANGULATION
2: objects/Polytope/properties/Triangulation and volume/TRIANGULATION
3: objects/Visualization/Visual::PointConfiguration/methods/TRIANGULATION
4: objects/Visualization/Visual::Polytope/methods/TRIANGULATION
5: objects/PointConfiguration/properties/Triangulation and volume/TRIANGULATION
    
```

By default, we just wait until polymake returns a result. However, it is possible to explicitly set a timeout. The following usually does work in an interactive session and often in doc tests, too. However, sometimes it hangs, and therefore we remove it from the tests, for now:

```

sage: c = polymake.cube(15)              # optional - jupymake
sage: polymake.eval('print {}->F_VECTOR;'.format(c.name()), timeout=1) # not
↳tested # optional - jupymake
Traceback (most recent call last):
...
RuntimeError: Polymake fails to respond timely
    
```

We verify that after the timeout, polymake is still able to give answers:

```

sage: c                                # optional - jupymake
cube of dimension 15
sage: c.N_VERTICES                      # optional - jupymake
32768
    
```

Note, however, that the recovery after a timeout is not perfect. It may happen that in some situation the interface collapses and thus polymake would automatically be restarted, thereby losing all data that have been computed before.

is_running()

Return True if self is currently running.

`sage.interfaces.polymake.polymake_console(command="")`

Spawn a new polymake command-line session.

EXAMPLES:

```

sage: from sage.interfaces.polymake import polymake_console
sage: polymake_console()          # not tested
Welcome to polymake version ...
...
Eugenij Gawrilow, Michael Joswig (TU Berlin)
http://www.polymake.org

This is free software licensed under GPL; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
↳PURPOSE.

Press F1 or enter 'help;' for basic instructions.

Application polytope currently uses following third-party software packages:
4ti2, bliss, cdd, latte, libnormaliz, lrs, permlib, ppl, sketch, sympol, threejs,
↳tikz, topcom, tosimplex
For more details: show_credits;
polytope >

```

`sage.interfaces.polymake.reduce_load_Polymake()`

Return the polymake interface object defined in `sage.interfaces.polymake`.

EXAMPLES:

```

sage: from sage.interfaces.polymake import reduce_load_Polymake
sage: reduce_load_Polymake()
Polymake

```

POV-RAY, THE PERSISTENCE OF VISION RAY TRACER

class sage.interfaces.povray.POVRay

Bases: object

POV-Ray The Persistence of Vision Ray Tracer

INPUT:

- `pov_file` – complete path to the .pov file you want to be rendered
- `outfile` – the filename you want to save your result to
- `**kwargs` – additionally keyword arguments you want to pass to POV-Ray

OUTPUT:

Image is written to the file you specified in outfile

EXAMPLES:

AUTHOR:

Sage interface written by Yi Qiang (yqiang_atNOSPAM_gmail.com)

POV-Ray: <http://www.povray.org>

usage()

PARALLEL INTERFACE TO THE SAGE INTERPRETER

This is an expect interface to `emph{multiple}` copy of the sage interpreter, which can all run simultaneous calculations. A PSage object does not work as well as the usual Sage object, but does have the great property that when you construct an object in a PSage you get back a prompt immediately. All objects constructed for that PSage print `<<currently executing code>>` until code execution completes, when they print as normal.

note{BUG – currently non-idle PSage subprocesses do not stop when sage exits. I would very much like to fix this but don't know how.}

EXAMPLES:

We illustrate how to factor 3 integers in parallel. First start up 3 parallel Sage interfaces:

```
sage: v = [PSage() for _ in range(3)]
```

Next, request factorization of one random integer in each copy.

```
sage: w = [x('factor(2^%s-1)% randint(250,310)) for x in v] # long time (5s on sage.
↳math, 2011)
```

Print the status:

```
sage: w # long time, random output (depends on timing)
[3 * 11 * 31^2 * 311 * 11161 * 11471 * 73471 * 715827883 * 2147483647 * 4649919401 *
↳18158209813151 * 5947603221397891 * 29126056043168521,
<<currently executing code>>,
9623 * 68492481833 *
↳23579543011798993222850893929565870383844167873851502677311057483194673]
```

Note that at the point when we printed two of the factorizations had finished but a third one hadn't. A few seconds later all three have finished:

```
sage: w # long time, random output
[3 * 11 * 31^2 * 311 * 11161 * 11471 * 73471 * 715827883 * 2147483647 * 4649919401 *
↳18158209813151 * 5947603221397891 * 29126056043168521,
23^2 * 47 * 89 * 178481 * 4103188409 * 199957736328435366769577 *
↳44667711762797798403039426178361,
9623 * 68492481833 *
↳23579543011798993222850893929565870383844167873851502677311057483194673]
```

```
class sage.interfaces.psage.PSage(**kwds)
```

```
    Bases: sage.interfaces.sage0.Sage
```

```
    eval(x, strip=True, **kwds)
```

```
        x – code strip –ignored
```

get(*var*)

Get the value of the variable *var*.

is_locked()

set(*var*, *value*)

Set the variable *var* to the given value.

class `sage.interfaces.psage.PSageElement`(*parent*, *value*, *is_name=False*, *name=None*)

Bases: `sage.interfaces.sage0.SageElement`

is_locked()

INTERFACE TO QEPCAD

The basic function of QEPCAD is to construct cylindrical algebraic decompositions (CADs) of \mathbf{R}^k , given a list of polynomials. Using this CAD, it is possible to perform quantifier elimination and formula simplification.

A CAD for a set A of k -variate polynomials decomposes \mathbf{R}^j into disjoint cells, for each j in $0 \leq j \leq k$. The sign of each polynomial in A is constant in each cell of \mathbf{R}^k , and for each cell in \mathbf{R}^j ($j > 1$), the projection of that cell into \mathbf{R}^{j-1} is a cell of \mathbf{R}^{j-1} . (This property makes the decomposition ‘cylindrical’.)

Given a formula $\exists x.P(a,b,x) = 0$ (for a polynomial P), and a cylindrical algebraic decomposition for P , we can eliminate the quantifier (find an equivalent formula in the two variables a, b without the quantifier \exists) as follows. For each cell C in \mathbf{R}^2 , find the cells of \mathbf{R}^3 which project to C . (This collection is called the `stack` over C .) Mark C as true if some member of the stack has `sign = 0`; otherwise, mark C as false. Then, construct a polynomial formula in a, b which specifies exactly the true cells (this is always possible). The same technique works if the body of the quantifier is any boolean combination of polynomial equalities and inequalities.

Formula simplification is a similar technique. Given a formula which describes a simple set of \mathbf{R}^k in a complicated way as a boolean combination of polynomial equalities and inequalities, QEPCAD can construct a CAD for the polynomials and recover a simple equivalent formula.

Note that while the following documentation is tutorial in nature, and is written for people who may not be familiar with QEPCAD, it is documentation for the sage interface rather than for QEPCAD. As such, it does not cover several issues that are very important to use QEPCAD efficiently, such as variable ordering, the efficient use of the alternate quantifiers and `_root_` expressions, the `measure-zero-error` command, etc. For more information on QEPCAD, see the online documentation at url{<http://www.cs.usna.edu/~qepcad/B/QEPCAD.html>} and Chris Brown’s tutorial handout and slides from url{<http://www.cs.usna.edu/~wcbrown/research/ISSAC04/Tutorial.html>}. (Several of the examples in this documentation came from these sources.)

The examples below require that the optional `qepcad` package is installed.

QEPCAD can be run in a fully automatic fashion, or interactively. We first demonstrate the automatic use of QEPCAD.

Since sage has no built-in support for quantifiers, this interface provides `qepcad_formula` which helps construct quantified formulas in the syntax QEPCAD requires.

```
sage: var('a,b,c,d,x,y,z')
(a, b, c, d, x, y, z)
sage: qf = qepcad_formula
```

We start with a simple example. Consider an arbitrarily-selected ellipse:

```
sage: ellipse = 3*x^2 + 2*x*y + y^2 - x + y - 7
```

What is the projection onto the x axis of this ellipse? First we construct a formula asking this question.

```
sage: F = qf.exists(y, ellipse == 0); F
(E y)[3 x^2 + 2 x y + y^2 - x + y - 7 = 0]
```

Then we run `qepcad` to get the answer:

```
sage: qepcad(F) # optional - qepcad
8 x^2 - 8 x - 29 <= 0
```

How about the projection onto the y axis?

```
sage: qepcad(qf.exists(x, ellipse == 0)) # optional - qepcad
8 y^2 + 16 y - 85 <= 0
```

QEPCAD deals with more quantifiers than just ‘exists’, of course. Besides the standard ‘forall’, there are also ‘for infinitely many’, ‘for all but finitely many’, ‘for a connected subset’, and ‘for exactly k ’. The `qepcad()` documentation has examples of all of these; here we will just give one example.

First we construct a circle:

```
sage: circle = x^2 + y^2 - 3
```

For what values k does a vertical line $x = k$ intersect the combined figure of the circle and ellipse exactly three times?

```
sage: F = qf.exactly_k(3, y, circle * ellipse == 0); F
(X3 y)[(3 x^2 + 2 x y + y^2 - x + y - 7) (x^2 + y^2 - 3) = 0]
sage: qepcad(F) # not tested (random order)
x^2 - 3 <= 0 /\ 8 x^2 - 8 x - 29 <= 0 /\ 8 x^4 - 26 x^2 - 4 x + 13 >= 0 /\ [ 8 x^4 - 26
↪ x^2 - 4 x + 13 = 0 \/ x^2 - 3 = 0 \/ 8 x^2 - 8 x - 29 = 0 ]
```

Here we see that the solutions are among the eight $(4 + 2 + 2)$ roots of the three polynomials inside the brackets, but not all of these roots are solutions; the polynomial inequalities outside the brackets are needed to select those roots that are solutions.

QEPCAD also supports an extended formula language, where `_root_k` $P(\bar{x}, y)$ refers to a particular zero of $P(\bar{x}, y)$ (viewed as a polynomial in y). If there are n roots, then `_root_1` refers to the least root and `_root_n` refers to the greatest. Also, `_root_-n` refers to the least root and `_root_-1` refers to the greatest.

This extended language is available both on input and output; see the QEPCAD documentation for more information on how to use this syntax on input. We can request output that is intended to be easy to interpret geometrically; then QEPCAD will use the extended language to produce a solution formula without the selection polynomials.

```
sage: qepcad(F, solution='geometric') # not tested (random order)
x = _root_1 8 x^2 - 8 x - 29
  \
  8 x^4 - 26 x^2 - 4 x + 13 = 0
  \
x = _root_-1 x^2 - 3
```

We then see that the 6 solutions correspond to the vertical tangent on the left side of the ellipse, the four intersections between the ellipse and the circle, and the vertical tangent on the right side of the circle.

Let us do some basic formula simplification and visualization. We will look at the region which is inside both the ellipse and the circle:

```
sage: F = qf.and_(ellipse < 0, circle < 0); F
[3 x^2 + 2 x y + y^2 - x + y - 7 < 0 /\ x^2 + y^2 - 3 < 0]
```

(continues on next page)

(continued from previous page)

```
sage: qepcad(F) # not tested (random order)
y^2 + 2 x y + y + 3 x^2 - x - 7 < 0 /\ y^2 + x^2 - 3 < 0
```

We get back the same formula we put in. This is not surprising (we started with a pretty simple formula, after all), but it is not very enlightening either. Again, if we ask for a ‘geometric’ output, then we see an output that lets us understand something about the shape of the solution set.

```
sage: qepcad(F, solution='geometric') # not tested (random order)
[
  [
    x = _root_-2 8 x^4 - 26 x^2 - 4 x + 13
    \
    x = _root_2 8 x^4 - 26 x^2 - 4 x + 13
    \
    8 x^4 - 26 x^2 - 4 x + 13 < 0
  ]
  /\
  y^2 + 2 x y + y + 3 x^2 - x - 7 < 0
  /\
  y^2 + x^2 - 3 < 0
]
\
[
  x > _root_2 8 x^4 - 26 x^2 - 4 x + 13
  /\
  x < _root_-2 8 x^4 - 26 x^2 - 4 x + 13
  /\
  y^2 + x^2 - 3 < 0
]
```

There is another reason to prefer output using `_root_` expressions; not only does it sometimes give added insight into the geometric structure, it also can be more efficient to construct. Consider this formula for the projection of a particular semicircle onto the x axis:

```
sage: F = qf.exists(y, qf.and_(circle == 0, x + y > 0)); F
(E y)[x^2 + y^2 - 3 = 0 /\ x + y > 0]
sage: qepcad(F) # not tested (random order)
x^2 - 3 <= 0 /\ [ x > 0 \ 2 x^2 - 3 < 0 ]
```

Here, the formula $x > 0$ had to be introduced in order to get a solution formula; the original CAD of F did not include the polynomial x . To avoid having QEPCAD do the extra work to come up with a solution formula, we can tell it to use the extended language; it is always possible to construct a solution formula in the extended language without introducing new polynomials.

```
sage: qepcad(F, solution='extended') # not tested (random order)
x^2 - 3 <= 0 /\ x > _root_1 2 x^2 - 3
```

Up to this point, all the output we have seen has basically been in the form of strings; there is no support (yet) for parsing these outputs back into sage polynomials (partly because sage does not yet have support for symbolic conjunctions and disjunctions). The function `qepcad()` supports three more output types that give numbers which can be manipulated in sage: any-point, all-points, and cell-points.

These output types give dictionaries mapping variable names to values. With any-point, `qepcad()` either produces a single dictionary specifying a point where the formula is true, or raises an exception if the formula is false everywhere.

(continued from previous page)

```
sage: intersections = pcombo(y=polygen(AA, 'y'), **pt); intersections # optional -_
↳ qepcad
y^4 + 4.464101615137755?*y^3 + 0.2679491924311227?*y^2
```

and verify that it does have three roots:

```
sage: intersections.roots() # optional -_
↳ qepcad
[(-4.403249005600958?, 1), (-0.06085260953679653?, 1), (0, 2)]
```

Let us check all six solutions.

```
sage: [len(pcombo(y=polygen(AA, 'y'), **p).roots()) for p in pts] # optional - qepcad
[3, 3, 3, 3, 3, 3]
```

We said earlier that we can run QEPCAD either automatically or interactively. Now that we have discussed the automatic modes, let us turn to interactive uses.

If the `qepcad()` function is passed `interact=True`, then instead of returning a result, it returns an object of class `Qepcad` representing a running instance of QEPCAD that you can interact with. For example:

```
sage: qe = qepcad(qf.forall(x, x^2 + b*x + c > 0), interact=True); qe # optional -_
↳ qepcad
QEPCAD object in phase 'Before Normalization'
```

This object is a fairly thin wrapper over QEPCAD; most QEPCAD commands are available as methods on the `Qepcad` object. Given a `Qepcad` object `qe`, you can type `qe.[tab]` to see the available QEPCAD commands; to see the documentation for an individual QEPCAD command, for example `d_setting`, you can type `qe.d_setting?`. (In QEPCAD, this command is called `d-setting`. We systematically replace hyphens with underscores for this interface.)

The execution of QEPCAD is divided into four phases. Most commands are not available during all phases. We saw above that QEPCAD starts out in phase 'Before Normalization'. We see that the `d_cell` command is not available in this phase:

```
sage: qe.d_cell() # optional - qepcad
Error GETCID: This command is not active here.
```

We will focus here on the fourth (and last) phase, 'Before Solution', because this interface has special support for some operations in this phase. Consult the QEPCAD documentation for information on the other phases.

We can tell QEPCAD to finish off the current phase and move to the next with its `go` command. (There is also the `step` command, which partially completes a phase for phases that have multiple steps, and the `finish` command, which runs QEPCAD to completion.)

```
sage: qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (x)'
```

```
sage: qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Choice'
```

```
sage: qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Solution'
```

Note that the `Qepcad` object returns the new phase whenever the phase changes, as a convenience for interactive use; except that when the new phase is 'EXITED', the solution formula printed by QEPCAD is returned instead.

```
sage: qe.go() # optional - qepcad
4 c - b^2 > 0
sage: qe # optional - qepcad
QEPCAD object in phase 'EXITED'
```

Let us pick a nice, simple example, return to phase 4, and explore the resulting qe object.

```
sage: qe = qepcad(circle == 0, interact=True); qe # optional - qepcad
QEPCAD object in phase 'Before Normalization'
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
```

We said before that QEPCAD creates ‘cylindrical algebraic decompositions’; since we have a bivariate polynomial, we get decompositions of \mathbf{R}^0 , \mathbf{R}^1 , and \mathbf{R}^2 . In this case, where our example is a circle of radius $\sqrt{3}$ centered on the origin, these decompositions are as follows:

The decomposition of \mathbf{R}^0 is trivial (of course). The decomposition of \mathbf{R}^1 has five cells: $x < -\sqrt{3}$, $x = -\sqrt{3}$, $-\sqrt{3} < x < \sqrt{3}$, $x = \sqrt{3}$, and $x > \sqrt{3}$. These five cells comprise the stack over the single cell in the trivial decomposition of \mathbf{R}^0 .

These five cells give rise to five stacks in \mathbf{R}^2 . The first and fifth stack have just one cell apiece. The second and fourth stacks have three cells: $y < 0$, $y = 0$, and $y > 0$. The third stack has five cells: below the circle, the lower semicircle, the interior of the circle, the upper semicircle, and above the circle.

QEPCAD (and this QEPCAD interface) number the cells in a stack starting with 1. Each cell has an index, which is a tuple of integers describing the path to the cell in the tree of all cells. For example, the cell ‘below the circle’ has index (3,1) (the first cell in the stack over the third cell of \mathbf{R}^1) and the interior of the circle has index (3,3).

We can view these cells with the QEPCAD command `d_cell`. For instance, let us look at the cell for the upper semicircle:

```
sage: qe.d_cell(3, 4) # optional - qepcad
----- Information about the cell (3,4) -----
Level : 2
Dimension : 1
Number of children : 0
Truth value : T by trial evaluation.
Degrees after substitution : Not known yet or No polynomial.
Multiplicities : ((1,1))
Signs of Projection Factors
Level 1 : (-)
Level 2 : (0)
----- Sample point -----
The sample point is in a PRIMITIVE representation.

alpha = the unique root of x^2 - 3 between 0 and 4
      = 1.7320508076-

Coordinate 1 = 0
             = 0.000000000000
Coordinate 2 = alpha
             = 1.7320508076-
-----
```

We see that, the level of this cell is 2, meaning that it is part of the decomposition of \mathbf{R}^2 . The dimension is 1, meaning that the cell is homeomorphic to a line (rather than a plane or a point). The sample point gives the coordinates of one point in the cell, both symbolically and numerically.

For programmatic access to cells, we have defined a sage wrapper class `QepcadCell`. These cells can be created with the `cell()` method; for example:

```
sage: c = qe.cell(3, 4); c                                     # optional - qepcad
QEPCAD cell (3, 4)
```

A `QepcadCell` has accessor methods for the important state held within a cell. For instance:

```
sage: c.level()                                             # optional - qepcad
2
sage: c.index()                                           # optional - qepcad
(3, 4)
sage: qe.cell(3).number_of_children()                     # optional - qepcad
5
sage: len(qe.cell(3))                                      # optional - qepcad
5
```

One particularly useful thing we can get from a cell is its sample point, as sage algebraic real numbers.

```
sage: c.sample_point()                                     # optional - qepcad
(0, 1.732050807568878?)
sage: c.sample_point_dict()                               # optional - qepcad
{'x': 0, 'y': 1.732050807568878?}
```

We have seen that we can get cells using the `cell()` method. There are several QEPCAD commands that print lists of cells; we can also get cells using the `make_cells()` method, passing it the output of one of these commands.

```
sage: qe.make_cells(qe.d_true_cells())                     # optional - qepcad
[QEPCAD cell (4, 2), QEPCAD cell (3, 4), QEPCAD cell (3, 2),
QEPCAD cell (2, 2)]
```

Also, the cells in the stack over a given cell can be accessed using array subscripting or iteration. (Remember that cells in a stack are numbered starting with one; we preserve this convention in the array-subscripting syntax.)

```
sage: c = qe.cell(3)                                       # optional - qepcad
sage: c[1]                                                 # optional - qepcad
QEPCAD cell (3, 1)
sage: [c2 for c2 in c]                                     # optional - qepcad
[QEPCAD cell (3, 1), QEPCAD cell (3, 2), QEPCAD cell (3, 3),
QEPCAD cell (3, 4), QEPCAD cell (3, 5)]
```

We can do one more thing with a cell: we can set its truth value. Once the truth values of the cells have been set, we can get QEPCAD to produce a formula which is true in exactly the cells we have selected. This is useful if QEPCAD's quantifier language is insufficient to express your problem.

For example, consider again our combined figure of the circle and the ellipse. Suppose you want to find all vertical lines that intersect the circle twice, and also intersect the ellipse twice. The vertical lines that intersect the circle twice can be found by simplifying:

```
sage: F = qf.exactly_k(2, y, circle == 0); F
(X2 y)[x^2 + y^2 - 3 = 0]
```

and the vertical lines that intersect the ellipse twice are expressed by:

```
sage: G = qf.exactly_k(2, y, ellipse == 0); G
(X2 y)[3 x^2 + 2 x y + y^2 - x + y - 7 = 0]
```

and the lines that intersect both figures would be:

```
sage: qf.and_(F, G)
Traceback (most recent call last):
...
ValueError: QEPCAD formulas must be in prenex (quantifiers outermost) form
```

...except that QEPCAD does not support formulas like this; in QEPCAD input, all logical connectives must be inside all quantifiers.

Instead, we can get QEPCAD to construct a CAD for our combined figure and set the truth values ourselves. (The exact formula we use doesn't matter, since we're going to replace the truth values in the cells; we just need to use a formula that uses both polynomials.)

```
sage: qe = qepcad(qf.and_(ellipse == 0, circle == 0), interact=True)      # optional - qe
→qepcad
sage: qe.go(); qe.go(); qe.go()          # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
```

Now we want to find all cells c in the decomposition of \mathbf{R}^1 such that the stack over c contains exactly two cells on the ellipse, and also contains exactly two cells on the circle.

Our input polynomials are 'level-2 projection factors', we see:

```
sage: qe.d_proj_factors()          # optional - qepcad
P_1,1 = fac(J_1,1) = fac(dis(A_2,1))
      = 8 x^2 - 8 x - 29
P_1,2 = fac(J_1,2) = fac(dis(A_2,2))
      = x^2 - 3
P_1,3 = fac(J_1,3) = fac(res(A_2,1|A_2,2))
      = 8 x^4 - 26 x^2 - 4 x + 13
A_2,1 = input
      = y^2 + 2 x y + y + 3 x^2 - x - 7
A_2,2 = input
      = y^2 + x^2 - 3
```

so we can test whether a cell is on the ellipse by checking that the sign of the corresponding projection factor is 0 in our cell. For instance, the cell (12,2) is on the ellipse:

```
sage: qe.cell(12,2).signs()[1][0]  # optional - qepcad
0
```

So we can update the truth values as desired like this:

```
sage: for c in qe.cell():          # optional - qepcad
.....:     count_ellipse = 0
.....:     count_circle = 0
.....:     for c2 in c:
```

(continues on next page)

(continued from previous page)

```
.....:     count_ellipse += (c2.signs()[1][0] == 0)
.....:     count_circle += (c2.signs()[1][1] == 0)
.....:     c.set_truth(count_ellipse == 2 and count_circle == 2)
```

and then we can get our desired solution formula. (The 'G' stands for 'geometric', and gives solutions using the same rules as `solution='geometric'` described above.)

```
sage: qe.solution_extension('G') # not tested (random order)
8 x^2 - 8 x - 29 < 0
^
x^2 - 3 < 0
```

AUTHORS:

- Carl Witty (2008-03): initial version
- Thierry Monteil (2015-07) repackaging + noncommutative doctests.

```
class sage.interfaces.qepcad.Qepcad(formula, vars=None, logfile=None, verbose=False, memcells=None,
server=None)
```

Bases: object

The wrapper for QEPCAD.

answer()

For a QEPCAD instance which is finished, return the simplified quantifier-free formula that it printed just before exiting.

EXAMPLES:

```
sage: qe = qepcad(x^3 - x == 0, interact=True) # optional - qepcad
sage: qe.finish() # not tested (random order)
x - 1 <= 0 /\ x + 1 >= 0 /\ [ x = 0 \/ x - 1 = 0 \/ x + 1 = 0 ]
sage: qe.answer() # not tested (random order)
x - 1 <= 0 /\ x + 1 >= 0 /\ [ x = 0 \/ x - 1 = 0 \/ x + 1 = 0 ]
```

assume(assume)

The following documentation is from `qepcad.help`.

Add an assumption to the problem. These will not be included in the solution formula.

For example, with input (E x)[a x^2 + b x + c = 0], if we issue the command

```
assume [ a != 0 ]
```

we will get the solution formula $b^2 - 4 a c \geq 0$. Without the assumption we'd get something like $[a = 0 / b \neq 0] / [a \neq 0 / 4 a c - b^2 \leq 0] / [a = 0 / b = 0 / c = 0]$.

EXAMPLES:

```
sage: var('a,b,c,x')
(a, b, c, x)
sage: qf = qepcad_formula
sage: qe = qepcad(qf.exists(x, a*x^2 + b*x + c == 0), interact=True) # optional - qepcad
↔ - qepcad
sage: qe.assume(a != 0) # optional - qepcad
sage: qe.finish() # optional - qepcad
4 a c - b^2 <= 0
```

cell(*index)

Given a cell index, returns a *QepcadCell* wrapper for that cell. Uses a cache for efficiency.

EXAMPLES:

```
sage: qe = qepcad(x + 3 == 42, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'At the end of projection phase'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.cell(2) # optional - qepcad
QEPCAD cell (2)
sage: qe.cell(2) is qe.cell(2) # optional - qepcad
True
```

final_stats()

For a QEPCAD instance which is finished, return the statistics that it printed just before exiting.

EXAMPLES:

```
sage: qe = qepcad(x == 0, interact=True) # optional - qepcad
sage: qe.finish() # optional - qepcad
x = 0
sage: qe.final_stats() # random, optional - qepcad
-----
0 Garbage collections, 0 Cells and 0 Arrays reclaimed, in 0 milliseconds.
492840 Cells in AVAIL, 500000 Cells in SPACE.
System time: 8 milliseconds.
System time after the initialization: 4 milliseconds.
-----
```

make_cells(text)

Given the result of some QEPCAD command that returns cells (such as *d_cell()*, *d_witness_list()*, etc.), return a list of cell objects.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: qe = qepcad(x^2 + y^2 == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.make_cells(qe.d_false_cells()) # optional - qepcad
[QEPCAD cell (5, 1), QEPCAD cell (4, 3), QEPCAD cell (4, 1), QEPCAD cell (3, 5),
↪ QEPCAD cell (3, 3), QEPCAD cell (3, 1), QEPCAD cell (2, 3), QEPCAD cell (2,
↪ 1), QEPCAD cell (1, 1)]
```

phase()

Return the current phase of the QEPCAD program.

EXAMPLES:

```
sage: qe = qepcad(x > 2/3, interact=True) # optional - qepcad
sage: qe.phase() # optional - qepcad
```

(continues on next page)

(continued from previous page)

```
'Before Normalization'
sage: qe.go() # optional - qepcad
QEPCAD object has moved to phase 'At the end of projection phase'
sage: qe.phase() # optional - qepcad
'At the end of projection phase'
sage: qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Choice'
sage: qe.phase() # optional - qepcad
'Before Choice'
sage: qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Solution'
sage: qe.phase() # optional - qepcad
'Before Solution'
sage: qe.go() # optional - qepcad
3 x - 2 > 0
sage: qe.phase() # optional - qepcad
'EXITED'
```

set_truth_value(index, nv)

Given a cell index (or a cell) and an integer, set the truth value of the cell to that integer.

Valid integers are 0 (false), 1 (true), and 2 (undetermined).

EXAMPLES:

```
sage: qe = qepcad(x == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'At the end of projection phase'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.set_truth_value(1, 1) # optional - qepcad
```

solution_extension(kind)

The following documentation is modified from `qepcad.help`:

solution-extension x

Use an alternative solution formula construction method. The parameter x is allowed to be T,E, or G. If x is T, then a formula in the usual language of Tarski formulas is produced. If x is E, a formula in the language of Extended Tarski formulas is produced. If x is G, then a geometry-based formula is produced.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: qf = qepcad_formula
sage: qe = qepcad(qf.and_(x^2 + y^2 - 3 == 0, x + y > 0), interact=True) #
↪ optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.solution_extension('E') # not tested (random order)
x > _root_1 2 x^2 - 3 /\ y^2 + x^2 - 3 = 0 /\ [ 2 x^2 - 3 > 0 \/ y = _root_-1 y^
↪ 2 + x^2 - 3 ]
```

(continues on next page)

(continued from previous page)

```

sage: qe.solution_extension('G') # not tested (random order)
[
  [
    2 x^2 - 3 < 0
    \
    x = _root_-1 2 x^2 - 3
  ]
  /\
  y = _root_-1 y^2 + x^2 - 3
]
\
[
  x^2 - 3 <= 0
  /\
  x > _root_-1 2 x^2 - 3
  /\
  y^2 + x^2 - 3 = 0
]
sage: qe.solution_extension('T') # not tested (random order)
y + x > 0 /\ y^2 + x^2 - 3 = 0

```

class sage.interfaces.qepcad.QepcadCell(parent, lines)

Bases: object

A wrapper for a QEPCAD cell.

index()

Give the index of a QEPCAD cell.

EXAMPLES:

```

sage: var('x,y')
(x, y)
sage: qe = qepcad(x^2 + y^2 == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.cell().index() # optional - qepcad
()
sage: qe.cell(1).index() # optional - qepcad
(1,)
sage: qe.cell(2, 2).index() # optional - qepcad
(2, 2)

```

level()

Return the level of a QEPCAD cell.

EXAMPLES:

```

sage: var('x,y')
(x, y)
sage: qe = qepcad(x^2 + y^2 == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad

```

(continues on next page)

(continued from previous page)

```

QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.cell().level() # optional - qepcad
0
sage: qe.cell(1).level() # optional - qepcad
1
sage: qe.cell(2, 2).level() # optional - qepcad
2

```

number_of_children()

Return the number of elements in the stack over a QEPCAD cell. (This is always an odd number, if the stack has been constructed.)

EXAMPLES:

```

sage: var('x,y')
(x, y)
sage: qe = qepcad(x^2 + y^2 == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.cell().number_of_children() # optional - qepcad
5
sage: [c.number_of_children() for c in qe.cell()] # optional - qepcad
[1, 3, 5, 3, 1]

```

sample_point()

Return the coordinates of a point in the cell, as a tuple of sage algebraic reals.

EXAMPLES:

```

sage: qe = qepcad(x^2 - x - 1 == 0, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'At the end of projection phase'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: v1 = qe.cell(2).sample_point()[0]; v1 # optional - qepcad
-0.618033988749895?
sage: v2 = qe.cell(4).sample_point()[0]; v2 # optional - qepcad
1.618033988749895?
sage: v1 + v2 == 1 # optional - qepcad
True

```

sample_point_dict()

Return the coordinates of a point in the cell, as a dictionary mapping variable names (as strings) to sage algebraic reals.

EXAMPLES:

```

sage: qe = qepcad(x^2 - x - 1 == 0, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'At the end of projection phase'

```

(continues on next page)

(continued from previous page)

```

QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.cell(4).sample_point_dict() # optional - qepcad
{'x': 1.618033988749895?}

```

set_truth(v)

Set the truth value of this cell, as used by QEPCAD for solution formula construction.

The argument *v* should be either a boolean or None (which will set the truth value to 'undetermined').

EXAMPLES:

```

sage: var('x,y')
(x, y)
sage: qe = qepcad(x^2 + y^2 == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: qe.solution_extension('T') # optional - qepcad
y^2 + x^2 - 1 = 0
sage: qe.cell(3, 3).set_truth(True) # optional - qepcad
sage: qe.solution_extension('T') # optional - qepcad
y^2 + x^2 - 1 <= 0

```

signs()

Return the sign vector of a QEPCAD cell.

This is a list of lists. The outer list contains one element for each level of the cell; the inner list contains one element for each projection factor at that level. These elements are either -1, 0, or 1.

EXAMPLES:

```

sage: var('x,y')
(x, y)
sage: qe = qepcad(x^2 + y^2 == 1, interact=True) # optional - qepcad
sage: qe.go(); qe.go(); qe.go() # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (y)'
QEPCAD object has moved to phase 'Before Choice'
QEPCAD object has moved to phase 'Before Solution'
sage: from sage.interfaces.qepcad import QepcadCell
sage: all_cells = flatten(qe.cell(), ltypes=QepcadCell, max_level=1) # optional - qepcad
↳ - qepcad
sage: [(c, c.signs()[1][0]) for c in all_cells] # optional - qepcad
[(QEPCAD cell (1, 1), 1), (QEPCAD cell (2, 1), 1), (QEPCAD cell (2, 2), 0),
↳ (QEPCAD cell (2, 3), 1), (QEPCAD cell (3, 1), 1), (QEPCAD cell (3, 2), 0),
↳ (QEPCAD cell (3, 3), -1), (QEPCAD cell (3, 4), 0), (QEPCAD cell (3, 5), 1),
↳ (QEPCAD cell (4, 1), 1), (QEPCAD cell (4, 2), 0), (QEPCAD cell (4, 3), 1),
↳ (QEPCAD cell (5, 1), 1)]

```

class sage.interfaces.qepcad.QepcadFunction(*parent, name*)

Bases: [sage.interfaces.expect.ExpectFunction](#)

A wrapper for a QEPCAD command.

```
class sage.interfaces.qepcad.Qepcad_expect(memcells=None, maxread=None, logfile=None,
                                          server=None)
```

Bases: `sage.interfaces.tab_completion.ExtraTabCompletion`, `sage.interfaces.expect.Expect`

The low-level wrapper for QEPCAD.

```
sage.interfaces.qepcad.qepcad(formula, assume=None, interact=False, solution=None, vars=None,
                               **kwargs)
```

Quantifier elimination and formula simplification using QEPCAD B.

If `assume` is specified, then the given formula is 'assumed', which is taken into account during final solution formula construction.

If `interact=True` is given, then a `Qepcad` object is returned which can be interacted with either at the command line or programmatically.

The type of solution returned can be adjusted with `solution`. The options are 'geometric', which tries to construct a solution formula with geometric meaning; 'extended', which gives a solution formula in an extended language that may be more efficient to construct; 'any-point', which returns any point where the formula is true; 'all-points', which returns a list of all points where the formula is true (or raises an exception if there are infinitely many); and 'cell-points', which returns one point in each cell where the formula is true.

All other keyword arguments are passed through to the `Qepcad` constructor.

For much more documentation and many more examples, see the module docstring for this module (type `sage.interfaces.qepcad?` to read this docstring from the sage command line).

The examples below require that the optional qepcad package is installed.

EXAMPLES:

```
sage: qf = qepcad_formula

sage: var('a,b,c,d,x,y,z,long_with_underscore_314159')
(a, b, c, d, x, y, z, long_with_underscore_314159)
sage: K.<q,r> = QQ[]

sage: qepcad('(E x)[a x + b > 0]', vars='(a,b,x)')      # not tested (random order)
a /= 0 ∨ b > 0

sage: qepcad(a > b)                                   # optional - qepcad
b - a < 0

sage: qepcad(qf.exists(x, a*x^2 + b*x + c == 0))      # not tested (random order)
4 a c - b^2 <= 0 ∧ [ c = 0 ∨ a /= 0 ∨ 4 a c - b^2 < 0 ]

sage: qepcad(qf.exists(x, a*x^2 + b*x + c == 0), assume=(a != 0))  # optional -
↪qepcad
4 a c - b^2 <= 0
```

For which values of a, b, c does $ax^2 + bx + c$ have 2 real zeroes?

```
sage: exact2 = qepcad(qf.exactly_k(2, x, a*x^2 + b*x + c == 0)); exact2  # not
↪tested (random order)
a /= 0 ∧ 4 a c - b^2 < 0
```

one real zero?

```
sage: exact1 = qepcad(qf.exactly_k(1, x, a*x^2 + b*x + c == 0)); exact1 # not tested
↳(random order)
[ a > 0 / \ 4 a c - b^2 = 0 ] \ \ [ a < 0 / \ 4 a c - b^2 = 0 ] \ \ [ a = 0 / \ 4 a c -
↳b^2 < 0 ]
```

No real zeroes?

```
sage: exact0 = qepcad(qf.forall(x, a*x^2 + b*x + c != 0)); exact0 # not tested
↳(random order)
4 a c - b^2 >= 0 / \ c /= 0 / \ [ b = 0 \ \ 4 a c - b^2 > 0 ]
```

3^{75} real zeroes?

```
sage: qepcad(qf.exactly_k(3^75, x, a*x^2 + b*x + c == 0)) # optional - qepcad
FALSE
```

We can check that the results don't overlap:

```
sage: qepcad(r'[[%s] / \ [%s]]' % (exact0, exact1), vars='a,b,c') # not tested
↳(random order)
FALSE
sage: qepcad(r'[[%s] / \ [%s]]' % (exact0, exact2), vars='a,b,c') # not tested
↳(random order)
FALSE
sage: qepcad(r'[[%s] / \ [%s]]' % (exact1, exact2), vars='a,b,c') # not tested
↳(random order)
FALSE
```

and that the union of the results is as expected:

```
sage: qepcad(r'[[%s] \ \ [%s] \ \ [%s]]' % (exact0, exact1, exact2), vars=(a,b,c)) #
↳not tested (random order)
b /= 0 \ \ a /= 0 \ \ c /= 0
```

So we have finitely many zeroes if a , b , or c is nonzero; which means we should have infinitely many zeroes if they are all zero.

```
sage: qepcad(qf.infinitely_many(x, a*x^2 + b*x + c == 0)) # not tested
↳(random order)
a = 0 / \ b = 0 / \ c = 0
```

The polynomial is nonzero almost everywhere iff it is not identically zero.

```
sage: qepcad(qf.all_but_finitely_many(x, a*x^2 + b*x + c != 0)) # not tested
↳(random order)
b /= 0 \ \ a /= 0 \ \ c /= 0
```

The non-zeroes are continuous iff there are no zeroes or if the polynomial is zero.

```
sage: qepcad(qf.connected_subset(x, a*x^2 + b*x + c != 0)) # not tested
↳(random order)
4 a c - b^2 >= 0 / \ [ a = 0 \ \ 4 a c - b^2 > 0 ]
```

The zeroes are continuous iff there are no or one zeroes, or if the polynomial is zero:

```

sage: qepcad(qf.connected_subset(x, a*x^2 + b*x + c == 0)) # not tested
↳(random order)
a = 0 \ / 4 a c - b^2 >= 0
sage: qepcad(r'[[%s] \ / [%s] \ / [a = 0 \ / b = 0 \ / c = 0]]' % (exact0, exact1),
↳vars='a,b,c') # not tested (random order)
a = 0 \ / 4 a c - b^2 >= 0

```

Since polynomials are continuous and $y > 0$ is an open set, they are positive infinitely often iff they are positive at least once.

```

sage: qepcad(qf.infinitely_many(x, a*x^2 + b*x + c > 0)) # not tested
↳(random order)
c > 0 \ / a > 0 \ / 4 a c - b^2 < 0
sage: qepcad(qf.exists(x, a*x^2 + b*x + c > 0)) # not tested
↳(random order)
c > 0 \ / a > 0 \ / 4 a c - b^2 < 0

```

However, since $y \geq 0$ is not open, the equivalence does not hold if you replace ‘positive’ with ‘nonnegative’. (We assume $a \neq 0$ to get simpler formulas.)

```

sage: qepcad(qf.infinitely_many(x, a*x^2 + b*x + c >= 0), assume=(a != 0)) # not
↳tested (random order)
a > 0 \ / 4 a c - b^2 < 0
sage: qepcad(qf.exists(x, a*x^2 + b*x + c >= 0), assume=(a != 0)) # not
↳tested (random order)
a > 0 \ / 4 a c - b^2 <= 0

```

sage.interfaces.qepcad.qepcad_banner()

Return the QEPCAD startup banner.

EXAMPLES:

```

sage: from sage.interfaces.qepcad import qepcad_banner
sage: qepcad_banner() # optional - qepcad
=====
           Quantifier Elimination
                in
Elementary Algebra and Geometry
                by
Partial Cylindrical Algebraic Decomposition
...
                by
                Hoon Hong
                (hhong@math.ncsu.edu)
With contributions by: Christopher W. Brown, George E.
Collins, Mark J. Encarnacion, Jeremy R. Johnson
Werner Krandick, Richard Liska, Scott McCallum,
Nicolas Robidoux, and Stanly Steinberg
=====

```

sage.interfaces.qepcad.qepcad_console(memcells=None)

Run QEPCAD directly. To exit early, press Control-C.

EXAMPLES:

```
sage: qepcad_console() # not tested
...
Enter an informal description between '[' and ']':
```

class sage.interfaces.qepcad.qepcad_formula_factory

Bases: object

Contains routines to help construct formulas in QEPCAD syntax.

A(*v, formula*)

Given a variable (or list of variables) and a formula, returns the universal quantification of the formula over the variables.

This method is available both as `forall()` and `A()` (the QEPCAD name for this quantifier).

The input formula may be a *qformula* as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.forall(a, a^2 + b > b^2 + a)
(A a)[a^2 + b > b^2 + a]
sage: qf.forall((a, b), a^2 + b^2 > 0)
(A a)(A b)[a^2 + b^2 > 0]
sage: qf.A(b, b^2 != a)
(A b)[b^2 /= a]
```

C(*v, formula, allow_multi=False*)

Given a variable and a formula, returns a new formula which is true iff the set of values for the variable at which the original formula was true is connected (including cases where this set is empty or is a single point).

This method is available both as `connected_subset()` and `C()` (the QEPCAD name for this quantifier).

The input formula may be a *qformula* as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.connected_subset(a, a^2 + b > b^2 + a)
(C a)[a^2 + b > b^2 + a]
sage: qf.C(b, b^2 != a)
(C b)[b^2 /= a]
```

E(*v, formula*)

Given a variable (or list of variables) and a formula, returns the existential quantification of the formula over the variables.

This method is available both as `exists()` and `E()` (the QEPCAD name for this quantifier).

The input formula may be a *qformula* as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.exists(a, a^2 + b > b^2 + a)
(E a)[a^2 + b > b^2 + a]
sage: qf.exists((a, b), a^2 + b^2 < 0)
(E a)(E b)[a^2 + b^2 < 0]
sage: qf.E(b, b^2 == a)
(E b)[b^2 = a]
```

F(v , *formula*)

Given a variable and a formula, returns a new formula which is true iff the original formula was true for infinitely many values of the variable.

This method is available both as *infinitely_many()* and *F()* (the QEPCAD name for this quantifier).

The input formula may be a *qformula* as returned by the methods of *qepcad_formula*, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.infinitely_many(a, a^2 + b > b^2 + a)
(F a)[a^2 + b > b^2 + a]
sage: qf.F(b, b^2 != a)
(F b)[b^2 /= a]
```

G(v , *formula*)

Given a variable and a formula, returns a new formula which is true iff the original formula was true for all but finitely many values of the variable.

This method is available both as *all_but_finitely_many()* and *G()* (the QEPCAD name for this quantifier).

The input formula may be a *qformula* as returned by the methods of *qepcad_formula*, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.all_but_finitely_many(a, a^2 + b > b^2 + a)
(G a)[a^2 + b > b^2 + a]
sage: qf.G(b, b^2 != a)
(G b)[b^2 /= a]
```

X(k , v , *formula*, *allow_multi=False*)

Given a nonnegative integer k , a variable, and a formula, returns a new formula which is true iff the original formula is true for exactly k values of the variable.

This method is available both as *exactly_k()* and *X()* (the QEPCAD name for this quantifier).

(Note that QEPCAD does not support $k = 0$ with this syntax, so if $k = 0$ is requested we implement it with *forall()* and *not_()*.)

The input formula may be a *qformula* as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.exactly_k(1, x, x^2 + a*x + b == 0)
(X1 x)[a x + x^2 + b = 0]
sage: qf.exactly_k(0, b, a*b == 1)
(A b)[~a b = 1]
```

`all_but_finitely_many(v, formula)`

Given a variable and a formula, returns a new formula which is true iff the original formula was true for all but finitely many values of the variable.

This method is available both as `all_but_finitely_many()` and `G()` (the QEPCAD name for this quantifier).

The input formula may be a *qformula* as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.all_but_finitely_many(a, a^2 + b > b^2 + a)
(G a)[a^2 + b > b^2 + a]
sage: qf.G(b, b^2 != a)
(G b)[b^2 /= a]
```

`and_(*formulas)`

Return the conjunction of its input formulas.

(This method would be named 'and' if that were not a Python keyword.)

Each input formula may be a *qformula* as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b,c,x')
(a, b, c, x)
sage: qf = qepcad_formula
sage: qf.and_(a*b, a*c, b*c != 0)
[a b = 0 /\ a c = 0 /\ b c /= 0]
sage: qf.and_(a*x^2 == 3, qf.or_(a > b, b > c))
[a x^2 = 3 /\ [a > b \/ b > c]]
```

`atomic(lhs, op='=', rhs=0)`

Construct a QEPCAD formula from the given inputs.

INPUT:

- *lhs* – a polynomial, or a symbolic equality or inequality
- *op* – a relational operator, default '='

- `rhs` – a polynomial, default 0

If `lhs` is a symbolic equality or inequality, then `op` and `rhs` are ignored.

This method works by printing the given polynomials, so we do not care what ring they are in (as long as they print with integral or rational coefficients).

EXAMPLES:

```
sage: qf = qepcad_formula
sage: var('a,b,c')
(a, b, c)
sage: K.<x,y> = QQ[]
sage: def test_qf(qf):
....:     return qf, qf.vars
sage: test_qf(qf.atomic(a^2 + 17))
(a^2 + 17 = 0, frozenset({'a'}))
sage: test_qf(qf.atomic(a*b*c <= c^3))
(a b c <= c^3, frozenset({'a', 'b', 'c'}))
sage: test_qf(qf.atomic(x+y^2, '!=', a+b))
(y^2 + x /= a + b, frozenset({'a', 'b', 'x', 'y'}))
sage: test_qf(qf.atomic(x, operator.lt))
(x < 0, frozenset({'x'}))
```

connected_subset(*v*, *formula*, *allow_multi=False*)

Given a variable and a formula, returns a new formula which is true iff the set of values for the variable at which the original formula was true is connected (including cases where this set is empty or is a single point).

This method is available both as `connected_subset()` and `C()` (the QEPCAD name for this quantifier).

The input formula may be a *qformula* as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.connected_subset(a, a^2 + b > b^2 + a)
(C a)[a^2 + b > b^2 + a]
sage: qf.C(b, b^2 != a)
(C b)[b^2 /= a]
```

exactly_k(*k*, *v*, *formula*, *allow_multi=False*)

Given a nonnegative integer k , a variable, and a formula, returns a new formula which is true iff the original formula is true for exactly k values of the variable.

This method is available both as `exactly_k()` and `X()` (the QEPCAD name for this quantifier).

(Note that QEPCAD does not support $k = 0$ with this syntax, so if $k = 0$ is requested we implement it with `forall()` and `not_()`.)

The input formula may be a *qformula* as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```

sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.exactly_k(1, x, x^2 + a*x + b == 0)
(X1 x)[a x + x^2 + b = 0]
sage: qf.exactly_k(0, b, a*b == 1)
(A b)[~a b = 1]

```

exists(*v*, *formula*)

Given a variable (or list of variables) and a formula, returns the existential quantification of the formula over the variables.

This method is available both as `exists()` and `E()` (the QEPCAD name for this quantifier).

The input formula may be a *qformula* as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial *p* (meaning $p = 0$).

EXAMPLES:

```

sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.exists(a, a^2 + b > b^2 + a)
(E a)[a^2 + b > b^2 + a]
sage: qf.exists((a, b), a^2 + b^2 < 0)
(E a)(E b)[a^2 + b^2 < 0]
sage: qf.E(b, b^2 == a)
(E b)[b^2 = a]

```

forall(*v*, *formula*)

Given a variable (or list of variables) and a formula, returns the universal quantification of the formula over the variables.

This method is available both as `forall()` and `A()` (the QEPCAD name for this quantifier).

The input formula may be a *qformula* as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial *p* (meaning $p = 0$).

EXAMPLES:

```

sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.forall(a, a^2 + b > b^2 + a)
(A a)[a^2 + b > b^2 + a]
sage: qf.forall((a, b), a^2 + b^2 > 0)
(A a)(A b)[a^2 + b^2 > 0]
sage: qf.A(b, b^2 != a)
(A b)[b^2 /= a]

```

formula(*formula*)

Constructs a QEPCAD formula from the given input.

INPUT:

- *formula* – a polynomial, a symbolic equality or inequality, or a list of polynomials, equalities, or inequalities

A polynomial p is interpreted as the equation $p = 0$. A list is interpreted as the conjunction ('and') of the elements.

EXAMPLES:

```
sage: var('a,b,c,x')
(a, b, c, x)
sage: qf = qepcad_formula
sage: qf.formula(a*x + b)
a x + b = 0
sage: qf.formula((a*x^2 + b*x + c, a != 0))
[a x^2 + b x + c = 0 /\ a != 0]
```

iff($f1, f2$)

Return the equivalence of its input formulas (that is, given formulas P and Q , returns ' P iff Q ').

The input formulas may be a *qformula* as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.iff(a, b)
[a = 0 <==> b = 0]
sage: qf.iff(a^2 < b, b^2 < a)
[a^2 < b <==> b^2 < a]
```

implies($f1, f2$)

Return the implication of its input formulas (that is, given formulas P and Q , returns ' P implies Q ').

The input formulas may be a *qformula* as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.implies(a, b)
[a = 0 ==> b = 0]
sage: qf.implies(a^2 < b, b^2 < a)
[a^2 < b ==> b^2 < a]
```

infinitely_many($v, formula$)

Given a variable and a formula, returns a new formula which is true iff the original formula was true for infinitely many values of the variable.

This method is available both as `infinitely_many()` and `F()` (the QEPCAD name for this quantifier).

The input formula may be a *qformula* as returned by the methods of `qepcad_formula`, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```
sage: var('a,b')
(a, b)
```

(continues on next page)

(continued from previous page)

```

sage: qf = qepcad_formula
sage: qf.infinitely_many(a, a^2 + b > b^2 + a)
(F a)[a^2 + b > b^2 + a]
sage: qf.F(b, b^2 != a)
(F b)[b^2 /= a]

```

not_(*formula*)

Return the negation of its input formula.

(This method would be named ‘not’ if that were not a Python keyword.)

The input formula may be a *qformula* as returned by the methods of *qepcad_formula*, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```

sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.not_(a > b)
[~a > b]
sage: qf.not_(a^2 + b^2)
[~a^2 + b^2 = 0]
sage: qf.not_(qf.and_(a > 0, b < 0))
[~[a > 0 /\ b < 0]]

```

or_(**formulas*)

Return the disjunction of its input formulas.

(This method would be named ‘or’ if that were not a Python keyword.)

Each input formula may be a *qformula* as returned by the methods of *qepcad_formula*, a symbolic equality or inequality, or a polynomial p (meaning $p = 0$).

EXAMPLES:

```

sage: var('a,b,c,x')
(a, b, c, x)
sage: qf = qepcad_formula
sage: qf.or_(a*b, a*c, b*c != 0)
[a b = 0 \/ a c = 0 \/ b c /= 0]
sage: qf.or_(a*x^2 == 3, qf.and_(a > b, b > c))
[a x^2 = 3 \/ [a > b /\ b > c]]

```

quantifier(*kind, v, formula, allow_multi=True*)

A helper method for building quantified QEPCAD formulas; not expected to be called directly.

Takes the quantifier kind (the string label of this quantifier), a variable or list of variables, and a formula, and returns the quantified formula.

EXAMPLES:

```

sage: var('a,b')
(a, b)
sage: qf = qepcad_formula
sage: qf.quantifier('NOT_A_REAL_QEPCAD_QUANTIFIER', a, a*b==0)

```

(continues on next page)

(continued from previous page)

```
(NOT_A_REAL_QEPCAD_QUANTIFIER a)[a b = 0]
sage: qf.quantifier('FOO', (a, b), a*b)
(FOO a)(FOO b)[a b = 0]
```

`sage.interfaces.qepcad.qepcad_version()`

Return a string containing the current QEPCAD version number.

EXAMPLES:

```
sage: qepcad_version() # random, optional - qepcad
'Version B 1.69, 16 Mar 2012'
```

class `sage.interfaces.qepcad.qformula`(*formula*, *vars*, *qvars=[]*)

Bases: `object`

A `qformula` holds a string describing a formula in QEPCAD's syntax, and a set of variables used.

INTERFACE TO BILL HART'S QUADRATIC SIEVE

`sage.interfaces.qsieve.data_to_list(out, n, time)`

Convert output of Hart's sieve and *n* to a list and time.

INPUT:

- *out* – snapshot of text output of Hart's QuadraticSieve program
- *n* – the integer being factored

OUTPUT:

- *list* – proper factors found so far
- *str* – time information

`sage.interfaces.qsieve.qsieve(n, block=True, time=False, verbose=False)`

Run Hart's quadratic sieve and return the distinct proper factors of the integer *n* that it finds.

CONDITIONS:

The conditions for the quadratic sieve to work are as follows:

- No small factors
- Not a perfect power
- Not prime

If any of these fails, the sieve will also.

INPUT:

- *n* – an integer with at least 40 digits
- *block* – (default: `True`) if `True`, you must wait until the sieve computation is complete before using Sage further. If `False`, Sage will run while the sieve computation runs in parallel. If *q* is the returned object, use `q.quit()` to terminate a running factorization.
- *time* – (default: `False`) if `True`, time the command using the UNIX “time” command (which you might have to install).
- *verbose* – (default: `False`) if `True`, print out verbose logging information about what happened during the Sieve run (for non-blocking Sieve, verbose information is always available via the `log()` method.)

OUTPUT:

- *list* – a list of the distinct proper factors of *n* found
- *str* – the time in cpu seconds that the computation took, as given by the command line `time` command. (If *time* is `False`, this is always an empty string.)

EXAMPLES:

```

sage: k = 19; n = next_prime(10^k)*next_prime(10^(k+1))
sage: factor(n) # (currently) uses PARI
10000000000000000000051 * 1000000000000000000039
sage: v, t = qsieve(n, time=True) # uses qsieve; optional - time
sage: v # optional - time
[10000000000000000000051, 1000000000000000000039]
sage: t # random; optional - time
'0.36 real      0.19 user      0.00 sys'

```

`sage.interfaces.qsieve.qsieve_block(n, time, verbose=False)`

Compute the factorization of `n` using Hart's quadratic Sieve blocking until complete.

class `sage.interfaces.qsieve.qsieve_nonblock(n, time)`

Bases: object

A non-blocking version of Hart's quadratic sieve.

The sieve starts running when you create the object, but you can still use Sage in parallel.

EXAMPLES:

```

sage: k = 19; n = next_prime(10^k)*next_prime(10^(k+1))
sage: q = qsieve(n, block=False, time=True) # optional - time
sage: q # random output; optional - time
Proper factors so far: []
sage: q # random output; optional - time
([10000000000000000000051, 1000000000000000000039], '0.21')
sage: q.list() # random output; optional - time
[10000000000000000000051, 1000000000000000000039]
sage: q.time() # random output; optional - time
'0.21'

sage: q = qsieve(next_prime(10^20)*next_prime(10^21), block=False)
sage: q # random output
Proper factors so far: [1000000000000000000039, 1000000000000000000117]
sage: q # random output
[1000000000000000000039, 1000000000000000000117]

```

`cputime()`

Return the time in seconds (as a string) that it took to factor `n`, or return “?” if the factorization has not completed or the time is unknown.

`done()`

Return True if the sieve process has completed.

`list()`

Return a list of the factors found so far, as Sage integers.

`log()`

Return all output of running the sieve so far.

`n()`

Return the integer that is being factored.

`pid()`

Return the PIN id of the QuadraticSieve process (actually of the time process that spawns the sieve process).

`quit()`

Terminate the QuadraticSieve process, in case you want to give up on computing this factorization.

EXAMPLES:

```
sage: n = next_prime(2^310)*next_prime(2^300)
sage: qs = qsieve(n, block=False)
sage: qs
Proper factors so far: []
sage: qs.quit()
sage: qs
Factorization was terminated early.
```

time()

Return the time in seconds (as a string) that it took to factor n , or return '?' if the factorization has not completed or the time is unknown.

INTERFACES TO R

This is the reference to the Sagemath R interface, usable from any Sage program.

The `%r` interface creating an R cell in the sage notebook is described in the Notebook manual.

The `%R` and `%%R` interface creating an R line or an R cell in the Jupyter notebook are briefly described at the end of this page. This documentation will be expanded and placed in the Jupyter notebook manual when this manual exists.

The following examples try to follow “An Introduction to R” which can be found at <http://cran.r-project.org/doc/manuals/R-intro.html>.

EXAMPLES:

Simple manipulations; numbers and vectors

The simplest data structure in R is the numeric vector which consists of an ordered collection of numbers. To create a vector named x using the R interface in Sage, you pass the R interpreter object a list or tuple of numbers:

```
sage: x = r([10.4,5.6,3.1,6.4,21.7]); x # optional - rpy2
[1] 10.4  5.6  3.1  6.4 21.7
```

You can invert elements of a vector x in R by using the invert operator or by doing $1/x$:

```
sage: ~x # optional - rpy2
[1] 0.09615385 0.17857143 0.32258065 0.15625000 0.04608295
sage: 1/x # optional - rpy2
[1] 0.09615385 0.17857143 0.32258065 0.15625000 0.04608295
```

The following assignment creates a vector y with 11 entries which consists of two copies of x with a 0 in between:

```
sage: y = r([x,0,x]); y # optional - rpy2
[1] 10.4  5.6  3.1  6.4 21.7  0.0 10.4  5.6  3.1  6.4 21.7
```

Vector Arithmetic

The following command generates a new vector v of length 11 constructed by adding together (element by element) $2x$ repeated 2.2 times, y repeated just once, and 1 repeated 11 times:

```
sage: v = 2*x+y+1; v # optional - rpy2
[1] 32.2 17.8 10.3 20.2 66.1 21.8 22.6 12.8 16.9 50.8 43.5
```

One can compute the sum of the elements of an R vector in the following two ways:

```
sage: sum(x) # optional - rpy2
[1] 47.2
```

(continues on next page)

(continued from previous page)

```
sage: x.sum() # optional - rpy2
[1] 47.2
```

One can calculate the sample variance of a list of numbers:

```
sage: ((x-x.mean())^2/(x.length()-1)).sum() # optional - rpy2
[1] 53.853
sage: x.var() # optional - rpy2
[1] 53.853

sage: x.sort() # optional - rpy2
[1] 3.1 5.6 6.4 10.4 21.7
sage: x.min() # optional - rpy2
[1] 3.1
sage: x.max() # optional - rpy2
[1] 21.7
sage: x # optional - rpy2
[1] 10.4 5.6 3.1 6.4 21.7

sage: r(-17).sqrt() # optional - rpy2
[1] NaN
sage: r('-17+0i').sqrt() # optional - rpy2
[1] 0+4.123106i
```

Generating an arithmetic sequence:

```
sage: r('1:10') # optional - rpy2
[1] 1 2 3 4 5 6 7 8 9 10
```

Because `from` is a keyword in Python, it can't be used as a keyword argument. Instead, `from_` can be passed, and R will recognize it as the correct thing:

```
sage: r.seq(length=10, from_=-1, by=.2) # optional - rpy2
[1] -1.0 -0.8 -0.6 -0.4 -0.2 0.0 0.2 0.4 0.6 0.8

sage: x = r([10.4,5.6,3.1,6.4,21.7]) # optional - rpy2
sage: x.rep(2) # optional - rpy2
[1] 10.4 5.6 3.1 6.4 21.7 10.4 5.6 3.1 6.4 21.7
sage: x.rep(times=2) # optional - rpy2
[1] 10.4 5.6 3.1 6.4 21.7 10.4 5.6 3.1 6.4 21.7
sage: x.rep(each=2) # optional - rpy2
[1] 10.4 10.4 5.6 5.6 3.1 3.1 6.4 6.4 21.7 21.7
```

Missing Values:

```
sage: na = r('NA') # optional - rpy2
sage: z = r([1,2,3,na]) # optional - rpy2
sage: z # optional - rpy2
[1] 1 2 3 NA
sage: ind = r.is_na(z) # optional - rpy2
sage: ind # optional - rpy2
[1] FALSE FALSE FALSE TRUE
sage: zero = r(0) # optional - rpy2
```

(continues on next page)

(continued from previous page)

```

sage: zero / zero # optional - rpy2
[1] NaN
sage: inf = r('Inf') # optional - rpy2
sage: inf-inf # optional - rpy2
[1] NaN
sage: r.is_na(inf) # optional - rpy2
[1] FALSE
sage: r.is_na(inf-inf) # optional - rpy2
[1] TRUE
sage: r.is_na(zero/zero) # optional - rpy2
[1] TRUE
sage: r.is_na(na) # optional - rpy2
[1] TRUE
sage: r.is_nan(inf-inf) # optional - rpy2
[1] TRUE
sage: r.is_nan(zero/zero) # optional - rpy2
[1] TRUE
sage: r.is_nan(na) # optional - rpy2
[1] FALSE

```

Character Vectors:

```

sage: labs = r.paste('c("X","Y")', '1:10', sep=''); labs # optional - rpy2
[1] "X1" "Y2" "X3" "Y4" "X5" "Y6" "X7" "Y8" "X9" "Y10"

```

Index vectors; selecting and modifying subsets of a data set:

```

sage: na = r('NA') # optional - rpy2
sage: x = r([10.4,5.6,3.1,6.4,21.7,na]); x # optional - rpy2
[1] 10.4 5.6 3.1 6.4 21.7 NA
sage: x['!is.na(self)'] # optional - rpy2
[1] 10.4 5.6 3.1 6.4 21.7

sage: x = r([10.4,5.6,3.1,6.4,21.7,na]); x # optional - rpy2
[1] 10.4 5.6 3.1 6.4 21.7 NA
sage: (x+1)[!is.na(self) & self>0] # optional - rpy2
[1] 11.4 6.6 4.1 7.4 22.7
sage: x = r([10.4,-2,3.1,-0.5,21.7,na]); x # optional - rpy2
[1] 10.4 -2.0 3.1 -0.5 21.7 NA
sage: (x+1)[!is.na(self) & self>0] # optional - rpy2
[1] 11.4 4.1 0.5 22.7

```

Distributions:

```

sage: r.options(width="60") # optional - rpy2
$width
[1] 80

sage: rr = r.dnorm(r.seq(-3,3,0.1)) # optional - rpy2
sage: rr # optional - rpy2
[1] 0.004431848 0.005952532 0.007915452 0.010420935
[5] 0.013582969 0.017528300 0.022394530 0.028327038
[9] 0.035474593 0.043983596 0.053990967 0.065615815

```

(continues on next page)

- The R interpreter launched by this interface IS (currently) DIFFERENT from the R interpreter used by other r... functions.
- Data exchanged via the -i and -o options have a format DIFFERENT from the format used by the r... functions (RPy2 mostly uses arrays, and bugs the user to use the pandas Python package).
- R graphics are (beautifully) displayed in output cells, but are not directly importable. You have to save them as .png, .pdf or .svg files and import them in Sage for further use.

In its current incarnation, this interface is mostly useful to statisticians needing Sage for a few symbolic computations but mostly using R for applied work.

AUTHORS:

- Mike Hansen (2007-11-01)
- William Stein (2008-04-19)
- Harald Schilly (2008-03-20)
- Mike Hansen (2008-04-19)
- Emmanuel Charpentier (2015-12-12, RPy2 interface)

class sage.interfaces.r.HelpExpression

Bases: str

Used to improve printing of output of r.help.

class sage.interfaces.r.R(maxread=None, logfile=None, init_list_length=1024, seed=None)

Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.interface.Interface

An interface to the R interpreter.

R is a comprehensive collection of methods for statistics, modelling, bioinformatics, data analysis and much more. For more details, see <http://www.r-project.org/about.html>

Resources:

- <http://r-project.org/> provides more information about R.
- <http://rseek.org/> R's own search engine.

EXAMPLES:

```
sage: r.summary(r.c(1,2,3,111,2,3,2,3,2,5,4)) # optional - rpy2
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
1.00   2.00   3.00  12.55   3.50  111.00
```

available_packages()

Returns a list of all available R package names.

This list is not necessarily sorted.

OUTPUT: list of strings

Note: This requires an internet connection. The CRAN server that is checked is defined at the top of sage/interfaces/r.py.

EXAMPLES:

```
sage: ap = r.available_packages() # optional - internet # optional - rpy2
sage: len(ap) > 20 # optional - internet # optional - rpy2
True
```

call(function_name, *args, **kwds)
 This is an alias for `function_call()`.

EXAMPLES:

```
sage: r.call('length', [1,2,3]) # optional - rpy2
[1] 3
```

chdir(dir)
 Changes the working directory to dir

INPUT:

- dir – the directory to change to.

EXAMPLES:

```
sage: import tempfile # optional - rpy2
sage: tmpdir = tempfile.mkdtemp() # optional - rpy2
sage: r.chdir(tmpdir) # optional - rpy2
```

Check that tmpdir and r.getwd() refer to the same directory. We need to use `realpath()` in case \$TMPDIR (by default /tmp) is a symbolic link (see trac ticket #10264).

```
sage: os.path.realpath(tmpdir) == sageobj(r.getwd()) # known bug (trac #9970)
↪ # optional - rpy2
True
```

completions(s)
 Return all commands names that complete the command starting with the string s. This is like typing s[Ctrl-T] in the R interpreter.

INPUT:

- s – string

OUTPUT: list – a list of strings

EXAMPLES:

```
sage: dummy = r._tab_completion(use_disk_cache=False) #clean doctest #_
↪ optional - rpy2
sage: 'testInheritedMethods' in r.completions('tes') # optional - rpy2
True
```

console()
 Runs the R console as a separate new R process.

EXAMPLES:

```
sage: r.console() # not tested # optional - rpy2
R version 2.6.1 (2007-11-26)
Copyright (C) 2007 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
...
```

convert_r_list(*l*)

Converts an R list to a Python list.

EXAMPLES:

```
sage: s = 'c(".GlobalEnv", "package:stats", "package:graphics",
↪ "package:grDevices", \n"package:utils", "package:datasets", "package:methods",
↪ "Autoloads", \n"package:base")' # optional - rpy2
sage: r.convert_r_list(s) # optional - rpy2
['.GlobalEnv',
 'package:stats',
 'package:graphics',
 'package:grDevices',
 'package:utils',
 'package:datasets',
 'package:methods',
 'Autoloads',
 'package:base']
```

eval(*code*, **args*, *kwds*)**

Evaluates a command inside the R interpreter and returns the output as a string.

EXAMPLES:

```
sage: r.eval('1+1') # optional - rpy2
'[1] 2'
```

function_call(*function*, *args*=None, *kwds*=None)

Return the result of calling an R function, with given args and keyword args.

OUTPUT: RElement – an object in R

EXAMPLES:

```
sage: r.function_call('length', args=[ [1,2,3] ]) # optional - rpy2
[1] 3
```

get(*var*)

Returns the string representation of the variable var.

INPUT:

- var – a string

OUTPUT: string

EXAMPLES:

```
sage: r.set('a', 2) # optional - rpy2
sage: r.get('a') # optional - rpy2
'[1] 2'
```

help(*command*)

Returns help string for a given command.

INPUT: - command – a string

OUTPUT: HelpExpression – a subclass of string whose `__repr__` method is `__str__`, so it prints nicely

EXAMPLES:

```
sage: r.help('c') # optional - rpy2
title
-----

Combine Values into a Vector or List

name
-----

c
...
```

install_packages(*package_name*)

Install an R package into Sage's R installation.

EXAMPLES:

```
sage: r.install_packages('aaMI') # not tested # optional - rpy2
...
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
...
Please restart Sage in order to use 'aaMI'.
```

library(*library_name*)

Load the library *library_name* into the R interpreter.

This function raises an ImportError if the given library is not known.

INPUT:

- *library_name* – string

EXAMPLES:

```
sage: r.library('grid') # optional - rpy2
sage: 'grid' in r.eval('(.packages())') # optional - rpy2
True
sage: r.library('foobar') # optional - rpy2
Traceback (most recent call last):
...
ImportError: ...
```

na()

Returns the NA in R.

OUTPUT: RElement – an element of R

EXAMPLES:

```
sage: r.na() # optional - rpy2
[1] NA
```

plot(*args, **kwds)

The R plot function. Type `r.help('plot')` for much more extensive documentation about this function. See also below for a brief introduction to more plotting with R.

If one simply wants to view an R graphic, using this function is sufficient (because it calls `dev.off()` to turn off the device).

However, if one wants to save the graphic to a specific file, it should be used as in the example below to write the output.

EXAMPLES:

This example saves a plot to the standard R output, usually a filename like `Rplot001.png` - from the command line, in the current directory, and in the cell directory in the notebook. We use a temporary directory in this example while doctesting this example, but you should use something persistent in your own code:

```
sage: from tempfile import TemporaryDirectory
sage: with TemporaryDirectory() as d: # optional - rpy2, rgraphics
.....:     _ = r.setwd(d)
.....:     r.plot("1:10")
null device
      1
```

To save to a specific file name, one should use `png()` to set the output device to that file. If this is done in the notebook, it must be done in the same cell as the plot itself:

```
sage: filename = tmp_filename() + '.png' # optional - rpy2
sage: r.png(filename="%s"%filename) # Note the double quotes in single quotes!
↪; optional -- rgraphics # optional - rpy2
NULL
sage: x = r([1,2,3]) # optional - rpy2
sage: y = r([4,5,6]) # optional - rpy2
sage: r.plot(x,y) # optional -- rgraphics # optional - rpy2
null device
      1
sage: import os; os.unlink(filename) # For doctesting, we remove the file;↪
↪optional -- rgraphics # optional - rpy2
```

Please note that for more extensive use of R's plotting capabilities (such as the `lattice` package), it is advisable to either use an interactive plotting device or to use the notebook. The following examples are not tested, because they differ depending on operating system:

```
sage: r.X11() # not tested - opens interactive device on systems with X11↪
↪support # optional - rpy2
sage: r.quartz() # not tested - opens interactive device on OSX # optional ↪
↪rpy2
sage: r.hist("rnorm(100)") # not tested - makes a plot # optional - rpy2
sage: r.library("lattice") # not tested - loads R lattice plotting package ↪
↪optional - rpy2
sage: r.histogram(x = "~ wt | cyl", data="mtcars") # not tested - makes a↪
↪lattice plot # optional - rpy2
sage: r.dev_off() # not tested, turns off the interactive viewer # optional ↪
↪rpy2
```

In the notebook, one can use `r.png()` to open the device, but would need to use the following since R `lattice` graphics do not automatically print away from the command line:

```
sage: filename = tmp_filename() + '.png' # Not needed in notebook, used for
↳doctesting # optional - rpy2
sage: r.png(filename="%s"%filename) # filename not needed in notebook, used
↳for doctesting; optional -- rgraphics # optional - rpy2
NULL
sage: r.library("lattice") # optional - rpy2
sage: r("print(histogram(~wt | cyl, data=mtcars))") # plot should appear;
↳optional -- rgraphics # optional - rpy2
sage: import os; os.unlink(filename) # We remove the file for doctesting, not
↳needed in notebook; optional -- rgraphics # optional - rpy2
```

png(*args, **kwargs)

Creates an R PNG device.

This should primarily be used to save an R graphic to a custom file. Note that when using this in the notebook, one must plot in the same cell that one creates the device. See `r.plot()` documentation for more information about plotting via R in Sage.

These examples won't work on the many platforms where R still gets built without graphics support.

EXAMPLES:

```
sage: filename = tmp_filename() + '.png' # optional - rpy2
sage: r.png(filename="%s"%filename) # optional -- rgraphics #
↳optional - rpy2
NULL
sage: x = r([1,2,3]) # optional - rpy2
sage: y = r([4,5,6]) # optional - rpy2
sage: r.plot(x,y) # This saves to filename, but is not viewable from command
↳line; optional -- rgraphics # optional - rpy2
null device
      1
sage: import os; os.unlink(filename) # We remove the file for doctesting;
↳optional -- rgraphics # optional - rpy2
```

We want to make sure that we actually can view R graphics, which happens differently on different platforms:

```
sage: s = r.eval('capabilities("png")') # Should be on Linux and Solaris #
↳optional - rpy2
sage: t = r.eval('capabilities("aqua")') # Should be on all supported Mac
↳versions # optional - rpy2
sage: "TRUE" in s+t # optional -- rgraphics # optional -
↳rpy2
True
```

read(filename)

Read filename into the R interpreter by calling R's source function on a read-only file connection.

EXAMPLES:

```
sage: filename = tmp_filename() # optional - rpy2
sage: f = open(filename, 'w') # optional - rpy2
sage: _ = f.write('a <- 2+2\n') # optional - rpy2
sage: f.close() # optional - rpy2
```

(continues on next page)

(continued from previous page)

```
sage: r.read(filename) # optional - rpy2
sage: r.get('a') # optional - rpy2
'[1] 4'
```

require(library_name)

Load the library library_name into the R interpreter.

This function raises an ImportError if the given library is not known.

INPUT:

- library_name – string

EXAMPLES:

```
sage: r.library('grid') # optional - rpy2
sage: 'grid' in r.eval('(.packages())') # optional - rpy2
True
sage: r.library('foobar') # optional - rpy2
Traceback (most recent call last):
...
ImportError: ...
```

set(var, value)

Set the variable var in R to what the string value evaluates to in R.

INPUT:

- var – a string
- value – a string

EXAMPLES:

```
sage: r.set('a', '2 + 3') # optional - rpy2
sage: r.get('a') # optional - rpy2
'[1] 5'
```

set_seed(seed=None)

Set the seed for R interpreter.

The seed should be an integer.

EXAMPLES:

```
sage: r = R() # optional - rpy2
sage: r.set_seed(1) # optional - rpy2
1
sage: r.sample("1:10", 5) # random # optional - rpy2
[1] 3 4 5 7 2
```

source(s)

Display the R source (if possible) about the function named s.

INPUT:

- s – a string representing the function whose source code you want to see

OUTPUT: string – source code

EXAMPLES:

```
sage: print(r.source("c")) # optional - rpy2
function (...) .Primitive("c")
```

version()

Return the version of R currently running.

OUTPUT: tuple of ints; string

EXAMPLES:

```
sage: r.version() # not tested # optional - rpy2
((3, 0, 1), 'R version 3.0.1 (2013-05-16)')
sage: rint, rstr = r.version() # optional - rpy2
sage: rint[0] >= 3 # optional - rpy2
True
sage: rstr.startswith('R version') # optional - rpy2
True
```

class sage.interfaces.r.**RElement**(parent, value, is_name=False, name=None)

Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.interface.InterfaceElement

dot_product(other)

Implements the notation self . other.

INPUT:

- self, other – R elements

OUTPUT: R element

EXAMPLES:

```
sage: c = r.c(1,2,3,4) # optional - rpy2
sage: c.dot_product(c.t()) # optional - rpy2
[,1] [,2] [,3] [,4]
[1,] 1 2 3 4
[2,] 2 4 6 8
[3,] 3 6 9 12
[4,] 4 8 12 16

sage: v = r([3,-1,8]) # optional - rpy2
sage: v.dot_product(v) # optional - rpy2
[,1]
[1,] 74
```

is_string()

Tell whether this element is a string.

EXAMPLES:

```
sage: r("abc").is_string() # optional - rpy2
True
sage: r([1,2,3]).is_string() # optional - rpy2
False
```

stat_model(x)

The tilde regression operator in R.

EXAMPLES:

```
sage: x = r([1,2,3,4,5]) # optional - rpy2
sage: y = r([3,5,7,9,11]) # optional - rpy2
sage: a = r.lm( y.tilde(x) ) # lm( y ~ x ) # optional - rpy2
sage: d = a._sage_() # optional - rpy2
sage: d['DATA']['coefficients']['DATA'][1] # optional - rpy2
2
```

tilde(x)

The tilde regression operator in R.

EXAMPLES:

```
sage: x = r([1,2,3,4,5]) # optional - rpy2
sage: y = r([3,5,7,9,11]) # optional - rpy2
sage: a = r.lm( y.tilde(x) ) # lm( y ~ x ) # optional - rpy2
sage: d = a._sage_() # optional - rpy2
sage: d['DATA']['coefficients']['DATA'][1] # optional - rpy2
2
```

class sage.interfaces.r.RFunction(parent, name, r_name=None)

Bases: *sage.interfaces.interface.InterfaceFunction*

A Function in the R interface.

INPUT:

- parent – the R interface
- name – the name of the function for Python
- r_name – the name of the function in R itself (which can have dots in it)

EXAMPLES:

```
sage: length = r.length # optional - rpy2
sage: type(length) # optional - rpy2
<class 'sage.interfaces.r.RFunction'>
sage: loads(dumps(length)) # optional - rpy2
length
```

class sage.interfaces.r.RFunctionElement(obj, name)

Bases: *sage.interfaces.interface.InterfaceFunctionElement*

sage.interfaces.r.is_RElement(x)

Return True if x is an element in an R interface.

INPUT:

- x – object

OUTPUT: bool

EXAMPLES:

```
sage: from sage.interfaces.r import is_RElement # optional - rpy2
sage: is_RElement(2) # optional - rpy2
False
sage: is_RElement(r(2)) # optional - rpy2
True
```

`sage.interfaces.r.r_console()`

Spawn a new R command-line session.

EXAMPLES:

```
sage: r_console() # not tested # optional - rpy2
R version 2.6.1 (2007-11-26)
Copyright (C) 2007 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
...
```

`sage.interfaces.r.r_version()`

Return the R version.

EXAMPLES:

```
sage: r_version() # not tested # optional - rpy2
((3, 0, 1), 'R version 3.0.1 (2013-05-16)')
sage: rint, rstr = r_version() # optional - rpy2
sage: rint[0] >= 3 # optional - rpy2
True
sage: rstr.startswith('R version') # optional - rpy2
True
```

`sage.interfaces.r.reduce_load_R()`

Used for reconstructing a copy of the R interpreter from a pickle.

EXAMPLES:

```
sage: from sage.interfaces.r import reduce_load_R # optional - rpy2
sage: reduce_load_R() # optional - rpy2
R Interpreter
```

INTERFACE TO SEVERAL RUBIK'S CUBE SOLVERS.

The first is by Michael Reid, and tries to find an optimal solution given the cube's state, and may take a long time. See http://www.math.ucf.edu/~reid/Rubik/optimal_solver.html

The second is by Eric Dietz, and uses a standard (?) algorithm to solve the cube one level at a time. It is extremely fast, but often returns a far from optimal solution. See <https://web.archive.org/web/20121212175710/http://www.wrongway.org/?rubiksource>

The third is by Dik Winter and implements Kociemba's algorithm which finds reasonable solutions relatively quickly, and if it is kept running will eventually find the optimal solution.

AUTHOR:

– Optimal was written by Michael Reid <reid@math.ucf.edu> (2004) – Cubex was written by Eric Dietz <root@wrongway.org> (2003) – Kociemba was written by Dik T. Winter <dik.winter@cw.nl> (1993) – Initial interface by Robert Bradshaw (2007-08)

class sage.interfaces.rubik.CubexSolver

Bases: object

format_cube(facets)

solve(facets)

EXAMPLES:

```
sage: from sage.interfaces.rubik import *           # optional - rubiks
sage: C = RubiksCube("R U")                         # optional - rubiks
sage: CubexSolver().solve(C.facets())              # optional - rubiks
'R U'
sage: C = RubiksCube("R U F L B D")                # optional - rubiks
sage: sol = CubexSolver().solve(C.facets()); sol   # optional - rubiks
"U' L' L' U L U' L U D L L D' L' D L' D' L D L' U' L D' L' U L' B' U' L' U B L_
↪D L D' U' L' U L B L B' L' U L U' L' F' L' F L' F L F' L' D' L' D D L D' B L B
↪' L B' L B F' L F F B' L F' B D' D' L D B' B' L' D' B U' U' L' B' D' F' F' L_
↪D F'"
sage: RubiksCube(sol) == C                          # optional - rubiks
True
sage: C = RubiksCube("R2 F'")                      # optional - rubiks
sage: CubexSolver().solve(C.facets())              # optional - rubiks
"R' R' F'"
sage: C = RubiksCube().scramble()                  # optional - rubiks
sage: sol = CubexSolver().solve(C.facets())        # optional - rubiks
sage: C == RubiksCube(sol)                         # optional - rubiks
True
```

class sage.interfaces.rubik.DikSolver

Bases: object

format_cube(facets)

solve(facets, timeout=10, extra_time=2)

EXAMPLES:

```
sage: from sage.interfaces.rubik import * # optional - rubiks
sage: C = RubiksCube().move("R U") # optional - rubiks
sage: DikSolver().solve(C.facets()) # optional - rubiks
'R U'
sage: C = RubiksCube().move("R U F L B D") # optional - rubiks
sage: DikSolver().solve(C.facets()) # optional - rubiks
'R U F L B D'
sage: C = RubiksCube().move("R2 F'") # optional - rubiks
sage: DikSolver().solve(C.facets()) # optional - rubiks
'R2 F''
```

class sage.interfaces.rubik.OptimalSolver(verbose=False, wait=True)

Bases: object

Interface to Michael Reid's optimal Rubik's Cube solver.

format_cube(facets)

ready()

solve(facets)

The initial startup and precomputation are substantial...

Todo: Let it keep searching once it found a solution?

EXAMPLES:

```
sage: from sage.interfaces.rubik import * # optional - rubiks
sage: solver = DikSolver() # optional - rubiks
sage: solver = OptimalSolver() # optional - rubiks # long time (28s on sage.
↳math, 2012)
Initializing tables...
Done.
sage: C = RubiksCube("R U") # optional - rubiks
sage: solver.solve(C.facets()) # optional - rubiks
'R U'
sage: C = RubiksCube("R U F L B D") # optional - rubiks
sage: solver.solve(C.facets()) # optional - rubiks
'R U F L B D'
sage: C = RubiksCube("R2 D2") # optional - rubiks
sage: solver.solve(C.facets()) # optional - rubiks
'R2 D2'
```

start()

stop()

class sage.interfaces.rubik.SingNot(s)

Bases: object

This class is to resolve difference between various Singmaster notation.

Case is ignored, and the second and third letters may be swapped.

EXAMPLES:

```
sage: from sage.interfaces.rubik import SingNot
sage: SingNot("acb") == SingNot("ACB")
True
sage: SingNot("acb") == SingNot("bca")
False
```


INTERFACE TO SAGE

This is an expect interface to *another* copy of the Sage interpreter.

```
class sage.interfaces.sage0.Sage(logfile=None, preparse=True, python=False, init_code=None,
                                server=None, server_tmpdir=None, remote_cleaner=True, **kwds)
Bases: sage.interfaces.tab_completion.ExtraTabCompletion, sage.interfaces.expect.Expect
```

Expect interface to the Sage interpreter itself.

INPUT:

- **server** - (optional); if specified runs Sage on a remote machine with address. You must have ssh keys setup so you can login to the remote machine by typing “ssh remote_machine” and no password, call `_install_hints_ssh()` for hints on how to do that.

The version of Sage should be the same as on the local machine, since pickling is used to move data between the two Sage process.

EXAMPLES: We create an interface to a copy of Sage. This copy of Sage runs as an external process with its own memory space, etc.

```
sage: s = Sage()
```

Create the element 2 in our new copy of Sage, and cube it.

```
sage: a = s(2)
sage: a^3
8
```

Create a vector space of dimension 4, and compute its generators:

```
sage: V = s('QQ^4')
sage: V.gens()
((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))
```

Note that `V` is not a vector space, it's a wrapper around an object (which happens to be a vector space), in another running instance of Sage.

```
sage: type(V)
<class 'sage.interfaces.sage0.SageElement'>
sage: V.parent()
Sage
sage: g = V.0; g
(1, 0, 0, 0)
```

(continues on next page)

(continued from previous page)

```
sage: g.parent()
Sage
```

We can still get the actual parent by using the name attribute of `g`, which is the variable name of the object in the child process.

```
sage: s('%s.parent()' % g.name())
Vector space of dimension 4 over Rational Field
```

Note that the memory space is completely different.

```
sage: x = 10
sage: s('x = 5')
5
sage: x
10
sage: s('x')
5
```

We can have the child interpreter itself make another child Sage process, so now three copies of Sage are running:

```
sage: s3 = s('Sage()')
sage: a = s3(10)
sage: a
10
```

This `a = 10` is in a subprocess of a subprocess of your original Sage.

```
sage: _ = s.eval('%s.eval("x=8")' % s3.name())
sage: s3('"x"')
8
sage: s('x')
5
sage: x
10
```

The double quotes are needed because the call to `s3` first evaluates its arguments using the `s` interpreter, so the call to `s3` is passed `s('"x"')`, which is the string `"x"` in the `s` interpreter.

`clear(var)`

Clear the variable named `var`.

Note that the exact format of the `NameError` for a cleared variable is slightly platform dependent, see [trac ticket #10539](#).

EXAMPLES:

```
sage: sage0.set('x', '2')
sage: sage0.get('x')
'2'
sage: sage0.clear('x')
sage: 'NameError' in sage0.get('x')
True
```

console()

Spawn a new Sage command-line session.

EXAMPLES:

```
sage: sage0.console() #not tested
-----
| SageMath version ..., Release Date: ... |
| Using Python .... Type "help()" for help. |
-----
...
```

cputime(*t=None*)

Return cputime since this Sage subprocess was started.

EXAMPLES:

```
sage: sage0.cputime() # random output
1.3530439999999999
sage: sage0('factor(2^157-1)')
852133201 * 60726444167 * 1654058017289 * 2134387368610417
sage: sage0.cputime() # random output
1.6462939999999999
```

eval(*line, strip=True, **kws*)

Send the code *x* to a second instance of the Sage interpreter and return the output as a string.

This allows you to run two completely independent copies of Sage at the same time in a unified way.

INPUT:

- *line* - input line of code
- *strip* - ignored

EXAMPLES:

```
sage: sage0.eval('2+2')
'4'
```

get(*var*)

Get the value of the variable *var*.

EXAMPLES:

```
sage: sage0.set('x', '2')
sage: sage0.get('x')
'2'
```

new(*x*)

EXAMPLES:

```
sage: sage0.new(2)
2
sage: _.parent()
Sage
```

preparse(*x*)

Returns the preparsed version of the string *s*.

EXAMPLES:

```
sage: sage0.preparse('2+2')
'Integer(2)+Integer(2)'
```

set(*var, value*)

Set the variable *var* to the given value.

EXAMPLES:

```
sage: sage0.set('x', '2')
sage: sage0.get('x')
'2'
```

version()

EXAMPLES:

```
sage: sage0.version()
'SageMath version ..., Release Date: ...'
sage: sage0.version() == version()
True
```

class `sage.interfaces.sage0.SageElement`(*parent, value, is_name=False, name=None*)

Bases: `sage.interfaces.expect.ExpectElement`

class `sage.interfaces.sage0.SageFunction`(*obj, name*)

Bases: `sage.interfaces.expect.FunctionElement`

`sage.interfaces.sage0.reduce_load_Sage`()

EXAMPLES:

```
sage: from sage.interfaces.sage0 import reduce_load_Sage
sage: reduce_load_Sage()
Sage
```

`sage.interfaces.sage0.reduce_load_element`(*s*)

EXAMPLES:

```
sage: from sage.interfaces.sage0 import reduce_load_element
sage: s = dumps(1/2)
sage: half = reduce_load_element(s); half
1/2
sage: half.parent()
Sage
```

`sage.interfaces.sage0.sage0_console`()

Spawn a new Sage command-line session.

EXAMPLES:

```
sage: sage0_console() #not tested
-----
| SageMath version ..., Release Date: ...           |
| Using Python ....   Type "help()" for help.       |
|-----|
...

```

sage.interfaces.sage0.sage0_version()

EXAMPLES:

```
sage: from sage.interfaces.sage0 import sage0_version
sage: sage0_version() == version()
True
```


INTERFACE TO SCILAB

Scilab is a scientific software package for numerical computations providing a powerful open computing environment for engineering and scientific applications. Scilab includes hundreds of mathematical functions with the possibility to add interactively programs from various languages (C, C++, Fortran...). It has sophisticated data structures (including lists, polynomials, rational functions, linear systems...), an interpreter and a high level programming language.

The commands in this section only work if you have the “scilab” interpreter installed and available in your PATH. It’s not necessary to install any special Sage packages.

EXAMPLES:

```
sage: scilab.eval('2+2')           # optional - scilab
'ans  =\n \n  4.'
sage: scilab('2+2')             # optional - scilab
4.
sage: a = scilab(10)            # optional - scilab
sage: a**10                     # optional - scilab
1.0000D+10
```

Tutorial based the MATLAB interface tutorial:

EXAMPLES:

```
sage: scilab('4+10')           # optional - scilab
14.
sage: scilab('date')           # optional - scilab; random output
15-Feb-2010
sage: scilab('5*10 + 6')       # optional - scilab
56.
sage: scilab('(6+6)/3')        # optional - scilab
4.
sage: scilab('9')^2           # optional - scilab
81.
sage: a = scilab(10); b = scilab(20); c = scilab(30)   # optional - scilab
sage: avg = (a+b+c)/3          # optional - scilab
sage: avg                      # optional - scilab
20.
sage: parent(avg)              # optional - scilab
Scilab

sage: my_scalar = scilab('3.1415') # optional - scilab
sage: my_scalar                # optional - scilab
3.1415
```

(continues on next page)

(continued from previous page)

```

sage: my_vector1 = scilab('[1,5,7]')      # optional - scilab
sage: my_vector1                          # optional - scilab
1.    5.    7.
sage: my_vector2 = scilab('[1;5;7]')    # optional - scilab
sage: my_vector2                          # optional - scilab
1.
5.
7.
sage: my_vector1 * my_vector2            # optional - scilab
75.

sage: row_vector1 = scilab('[1 2 3]')    # optional - scilab
sage: row_vector2 = scilab('[3 2 1]')    # optional - scilab
sage: matrix_from_row_vec = scilab('[%s; %s]'%(row_vector1.name(), row_vector2.name()))
↳ # optional - scilab
sage: matrix_from_row_vec                # optional - scilab
1.    2.    3.
3.    2.    1.

sage: column_vector1 = scilab('[1;3]')   # optional - scilab
sage: column_vector2 = scilab('[2;8]')   # optional - scilab
sage: matrix_from_col_vec = scilab('[%s %s]%(column_vector1.name(), column_vector2.
↳name())') # optional - scilab
sage: matrix_from_col_vec                # optional - scilab
1.    2.
3.    8.

sage: my_matrix = scilab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]') # optional -
↳scilab
sage: my_matrix                          # optional - scilab
    8.    12.    19.
    7.     3.     2.
   12.    4.    23.
    8.     1.     1.

sage: combined_matrix = scilab('[%s, %s]%(my_matrix.name(), my_matrix.name())')
↳ # optional - scilab
sage: combined_matrix                    # optional - scilab
    8.    12.    19.    8.    12.    19.
    7.     3.     2.     7.     3.     2.
   12.    4.    23.   12.    4.    23.
    8.     1.     1.     8.     1.     1.

sage: tm = scilab('0.5:2:10')            # optional - scilab
sage: tm                                # optional - scilab
0.5    2.5    4.5    6.5    8.5

sage: my_vector1 = scilab('[1,5,7]')    # optional - scilab
sage: my_vector1(1)                    # optional - scilab
1.
sage: my_vector1(2)                    # optional - scilab
5.

```

(continues on next page)

(continued from previous page)

```
sage: my_vector1(3) # optional - scilab
7.
```

Matrix indexing works as follows:

```
sage: my_matrix = scilab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]') # optional - scilab
↪scilab
sage: my_matrix(3,2) # optional - scilab
4.
```

One can also use square brackets:

```
sage: my_matrix[3,2] # optional - scilab
4.
```

Setting using parenthesis cannot work (because of how the Python language works). Use square brackets or the set function:

```
sage: my_matrix = scilab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]') # optional - scilab
↪scilab
sage: my_matrix.set(2,3, 1999) # optional - scilab
sage: my_matrix # optional - scilab
  8.      12.      19.
  7.       3.     1999.
 12.       4.      23.
  8.       1.       1.
sage: my_matrix[2,3] = -126 # optional - scilab
sage: my_matrix # optional - scilab
  8.      12.      19.
  7.       3.     - 126.
 12.       4.      23.
  8.       1.       1.
```

AUTHORS:

– Ronan Paixao (2008-11-26), based on the MATLAB tutorial by William Stein (2006-10-11)

```
class sage.interfaces.scilab.Scilab(maxread=None, script_subdirectory=None, logfile=None,
server=None, server_tmpdir=None, seed=None)
```

Bases: *sage.interfaces.expect.Expect*

Interface to the Scilab interpreter.

EXAMPLES:

```
sage: a = scilab('[ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]') # optional - scilab
sage: b = scilab('[ 1; 3; 13]') # optional - scilab
sage: c = a * b # optional - scilab
sage: print(c) # optional - scilab
 30.
 122.
 505.
```

console()

Starts Scilab console.

EXAMPLES:

```
sage: scilab.console()           # optional - scilab; not tested
```

eval(*command*, *args, **kwds)

Evaluates commands.

EXAMPLES:

```
sage: scilab.eval("5")           # optional - scilab
'ans =
```

```
5.' sage: scilab.eval("d=44") # optional - scilab 'd =
```

```
44.'
```

get(*var*)

Get the value of the variable var.

EXAMPLES:

```
sage: scilab.eval('b=124;')     # optional - scilab
''
sage: scilab.get('b')           # optional - scilab
'
```

```
124.'
```

sage2scilab_matrix_string(*A*)

Return a Scilab matrix from a Sage matrix.

INPUT: A Sage matrix with entries in the rationals or reals.

OUTPUT: A string that evaluates to an Scilab matrix.

EXAMPLES:

```
sage: M33 = MatrixSpace(QQ,3,3) # optional - scilab
sage: A = M33([1,2,3,4,5,6,7,8,0]) # optional - scilab
sage: scilab.sage2scilab_matrix_string(A) # optional - scilab
'[1, 2, 3; 4, 5, 6; 7, 8, 0]'
```

set(*var*, *value*)

Set the variable var to the given value.

EXAMPLES:

```
sage: scilab.set('a', 123)     # optional - scilab
sage: scilab.get('a')          # optional - scilab
'
```

```
123.'
```

set_seed(*seed=None*)

Set the seed for gp interpreter.

The seed should be an integer.

EXAMPLES:

```
sage: from sage.interfaces.scilab import Scilab # optional - scilab
sage: s = Scilab() # optional - scilab
sage: s.set_seed(1) # optional - scilab
1
sage: [s.rand() for i in range(5)] # optional - scilab
[
    0.6040239,
    0.0079647,
    0.6643966,
    0.9832111,
    0.5321420]
```

version()

Returns the version of the Scilab software used.

EXAMPLES:

```
sage: scilab.version() # optional - scilab
'scilab-...'
```

whos(*name=None, typ=None*)

Returns information about current objects. Arguments: *nam*: first characters of selected names *typ*: name of selected Scilab variable type

EXAMPLES:

```
sage: scilab.whos("core") # optional - scilab
'Name          Type          Size          Bytes...'
sage: scilab.whos(typ='function') # optional - scilab
'Name          Type          Size          Bytes...'
```

class `sage.interfaces.scilab.ScilabElement`(*parent, value, is_name=False, name=None*)

Bases: `sage.interfaces.expect.ExpectElement`

set(*i, j, x*)

Set the variable *var* to the given value.

EXAMPLES:

```
sage: scilab.set('c', 125) # optional - scilab
sage: scilab.get('c') # optional - scilab
125
```

125.'

`sage.interfaces.scilab.scilab_console()`

This requires that the optional Scilab program be installed and in your PATH, but no optional Sage packages need to be installed.

EXAMPLES:

```
sage: from sage.interfaces.scilab import scilab_console # optional - scilab
sage: scilab_console()                               # optional - scilab; not tested

-----
                    scilab-5.0.3

                    Consortium Scilab (DIGITEO)
                    Copyright (c) 1989-2008 (INRIA)
                    Copyright (c) 1989-2007 (ENPC)
                    -----

Startup execution:
  loading initial environment

-->2+3
ans =

   5.

-->quit
```

Typing quit exits the Scilab console and returns you to Sage. Scilab, like Sage, remembers its history from one session to another.

`sage.interfaces.scilab.scilab_version()`

Return the version of Scilab installed.

EXAMPLES:

```
sage: from sage.interfaces.scilab import scilab_version # optional - scilab
sage: scilab_version()   # optional - scilab
'scilab-...'
```

INTERFACE TO SINGULAR

44.1 Introduction

This interface is extremely flexible, since it's exactly like typing into the Singular interpreter, and anything that works there should work here.

The Singular interface will only work if Singular is installed on your computer; this should be the case, since Singular is included with Sage. The interface offers three pieces of functionality:

1. `singular_console()` - A function that dumps you into an interactive command-line Singular session.
2. `singular(expr, type='def')` - Creation of a Singular object. This provides a Pythonic interface to Singular. For example, if `f=singular(10)`, then `f.factorize()` returns the factorization of 10 computed using Singular.
3. `singular.eval(expr)` - Evaluation of arbitrary Singular expressions, with the result returned as a string.

Of course, there are polynomial rings and ideals in Sage as well (often based on a C-library interface to Singular). One can convert an object in the Singular interpreter interface to Sage by the method `sage()`.

44.2 Tutorial

EXAMPLES: First we illustrate multivariate polynomial factorization:

```
sage: R1 = singular.ring(0, '(x,y)', 'dp')
sage: R1
polynomial ring, over a field, global ordering
// coefficients: QQ
// number of vars : 2
//      block  1 : ordering dp
//              : names    x y
//      block  2 : ordering C
sage: f = singular('9x16 - 18x13y2 - 9x12y3 + 9x10y4 - 18x11y2 + 36x8y4 + 18x7y5 -
↳18x5y6 + 9x6y4 - 18x3y6 - 9x2y7 + 9y8')
sage: f
9*x^16-18*x^13*y^2-9*x^12*y^3+9*x^10*y^4-18*x^11*y^2+36*x^8*y^4+18*x^7*y^5-18*x^5*y^
↳6+9*x^6*y^4-18*x^3*y^6-9*x^2*y^7+9*y^8
sage: f.parent()
Singular
```

```
sage: F = f.factorize(); F
[1]:
  _[1]=9
  _[2]=x^6-2*x^3*y^2-x^2*y^3+y^4
  _[3]=-x^5+y^2
[2]:
  1,1,2
```

```
sage: F[1]
9,
x^6-2*x^3*y^2-x^2*y^3+y^4,
-x^5+y^2
sage: F[1][2]
x^6-2*x^3*y^2-x^2*y^3+y^4
```

We can convert f and each exponent back to Sage objects as well.

```
sage: g = f.sage(); g
9*x^16 - 18*x^13*y^2 - 9*x^12*y^3 + 9*x^10*y^4 - 18*x^11*y^2 + 36*x^8*y^4 + 18*x^7*y^5 -
↳ 18*x^5*y^6 + 9*x^6*y^4 - 18*x^3*y^6 - 9*x^2*y^7 + 9*y^8
sage: F[1][2].sage()
x^6 - 2*x^3*y^2 - x^2*y^3 + y^4
sage: g.parent()
Multivariate Polynomial Ring in x, y over Rational Field
```

This example illustrates polynomial GCD's:

```
sage: R2 = singular.ring(0, '(x,y,z)', 'lp')
sage: a = singular.new('3x2*(x+y)')
sage: b = singular.new('9x*(y2-x2)')
sage: g = a.gcd(b)
sage: g
x^2+x*y
```

This example illustrates computation of a Groebner basis:

```
sage: R3 = singular.ring(0, '(a,b,c,d)', 'lp')
sage: I = singular.ideal(['a + b + c + d', 'a*b + a*d + b*c + c*d', 'a*b*c + a*b*d +
↳ a*c*d + b*c*d', 'a*b*c*d - 1'])
sage: I2 = I.groebner()
sage: I2
c^2*d^6-c^2*d^2-d^4+1,
c^3*d^2+c^2*d^3-c-d,
b*d^4-b+d^5-d,
b*c-b*d^5+c^2*d^4+c*d-d^6-d^2,
b^2+2*b*d+d^2,
a+b+c+d
```

The following example is the same as the one in the Singular - Gap interface documentation:

```
sage: R = singular.ring(0, '(x0,x1,x2)', 'lp')
sage: I1 = singular.ideal(['x0*x1*x2 - x0^2*x2', 'x0^2*x1*x2-x0*x1^2*x2-x0*x1*x2^2',
↳ 'x0*x1-x0*x2-x1*x2'])
```

(continues on next page)

(continued from previous page)

```

sage: I2 = I1.groebner()
sage: I2
x1^2*x2^2,
x0*x2^3-x1^2*x2^2+x1*x2^3,
x0*x1-x0*x2-x1*x2,
x0^2*x2-x0*x2^2-x1*x2^2
sage: I2.sage()
Ideal (x1^2*x2^2, x0*x2^3 - x1^2*x2^2 + x1*x2^3, x0*x1 - x0*x2 - x1*x2, x0^2*x2 - x0*x2^2
↪ - x1*x2^2) of Multivariate Polynomial Ring in x0, x1, x2 over Rational Field

```

This example illustrates moving a polynomial from one ring to another. It also illustrates calling a method of an object with an argument.

```

sage: R = singular.ring(0, '(x,y,z)', 'dp')
sage: f = singular('x3+y3+(x-y)*x2y2+z2')
sage: f
x^3*y^2-x^2*y^3+x^3+y^3+z^2
sage: R1 = singular.ring(0, '(x,y,z)', 'ds')
sage: f = R.fetch(f)
sage: f
z^2+x^3+y^3+x^3*y^2-x^2*y^3

```

We can calculate the Milnor number of f :

```

sage: _=singular.LIB('sing.lib')    # assign to _ to suppress printing
sage: f.milnor()
4

```

The Jacobian applied twice yields the Hessian matrix of f , with which we can compute.

```

sage: H = f.jacob().jacob()
sage: H
6*x+6*x*y^2-2*y^3, 6*x^2*y-6*x*y^2, 0,
6*x^2*y-6*x*y^2, 6*y+2*x^3-6*x^2*y, 0,
0, 0, 2
sage: H.sage()
[6*x + 6*x*y^2 - 2*y^3      6*x^2*y - 6*x*y^2      0]
[ 6*x^2*y - 6*x*y^2  6*y + 2*x^3 - 6*x^2*y      0]
[ 0 0 2]
sage: H.det()    # This is a polynomial in Singular
72*x*y+24*x^4-72*x^3*y+72*x*y^3-24*y^4-48*x^4*y^2+64*x^3*y^3-48*x^2*y^4
sage: H.det().sage() # This is the corresponding polynomial in Sage
72*x*y + 24*x^4 - 72*x^3*y + 72*x*y^3 - 24*y^4 - 48*x^4*y^2 + 64*x^3*y^3 - 48*x^2*y^4

```

The 1x1 and 2x2 minors:

```

sage: H.minor(1)
2,
6*y+2*x^3-6*x^2*y,
6*x^2*y-6*x*y^2,
6*x^2*y-6*x*y^2,
6*x+6*x*y^2-2*y^3,
0,

```

(continues on next page)

(continued from previous page)

```
0,
0,
0
sage: H.minor(2)
12*y+4*x^3-12*x^2*y,
12*x^2*y-12*x*y^2,
12*x^2*y-12*x*y^2,
12*x+12*x*y^2-4*y^3,
-36*x*y-12*x^4+36*x^3*y-36*x*y^3+12*y^4+24*x^4*y^2-32*x^3*y^3+24*x^2*y^4,
0,
0,
0,
0
```

```
sage: -=singular.eval('option(redSB)')
sage: H.minor(1).groebner()
1
```

44.3 Computing the Genus

We compute the projective genus of ideals that define curves over \mathbb{Q} . It is *very important* to load the `normal.lib` library before calling the `genus` command, or you'll get an error message.

EXAMPLES:

```
sage: singular.lib('normal.lib')
sage: R = singular.ring(0, '(x,y)', 'dp')
sage: i2 = singular.ideal('y^9 - x^2*(x-1)^9 + x')
sage: i2.genus()
40
```

Note that the genus can be much smaller than the degree:

```
sage: i = singular.ideal('y^9 - x^2*(x-1)^9')
sage: i.genus()
0
```

44.4 An Important Concept

The following illustrates an important concept: how Sage interacts with the data being used and returned by Singular. Let's compute a Groebner basis for some ideal, using Singular through Sage.

```
sage: singular.lib('polylib.lib')
sage: singular.ring(32003, '(a,b,c,d,e,f)', 'lp')
      polynomial ring, over a field, global ordering
      // coefficients: ZZ/32003
      // number of vars : 6
      //          block 1 : ordering lp
```

(continues on next page)

(continued from previous page)

```

//           : names  a b c d e f
//      block 2 : ordering C
sage: I = singular.ideal('cyclic(6)')
sage: g = singular('groebner(I)')
Traceback (most recent call last):
...
TypeError: Singular error:
...

```

We restart everything and try again, but correctly.

```

sage: singular.quit()
sage: singular.lib('polylib.lib'); R = singular.ring(32003, '(a,b,c,d,e,f)', 'lp')
sage: I = singular.ideal('cyclic(6)')
sage: I.groebner()
f^48-2554*f^42-15674*f^36+12326*f^30-12326*f^18+15674*f^12+2554*f^6-1,
...

```

It's important to understand why the first attempt at computing a basis failed. The line where we gave singular the input 'groebner(I)' was useless because Singular has no idea what 'I' is! Although 'I' is an object that we computed with calls to Singular functions, it actually lives in Sage. As a consequence, the name 'I' means nothing to Singular. When we called `I.groebner()`, Sage was able to call the `groebner` function on 'I' in Singular, since 'I' actually means something to Sage.

44.5 Long Input

The Singular interface reads in even very long input (using files) in a robust manner, as long as you are creating a new object.

```

sage: t = "%s"%10^15000 # 15 thousand character string (note that normal Singular_
↪input must be at most 10000)
sage: a = singular.eval(t)
sage: a = singular(t)

```

AUTHORS:

- David Joyner and William Stein (2005): first version
- Neal Harris (unknown): perhaps added “An Important Concept”
- Martin Albrecht (2006-03-05): code so `singular.[tab]` and `x = singular(...)`, `x.[tab]` includes all singular commands.
- Martin Albrecht (2006-03-06): This patch adds the equality symbol to `singular`. Also fix a problem in which `””` as prompt means comparison will break all further communication with Singular.
- Martin Albrecht (2006-03-13): added `current_ring()` and `current_ring_name()`
- William Stein (2006-04-10): Fixed problems with ideal constructor
- Martin Albrecht (2006-05-18): added `sage_poly`.
- Simon King (2010-11-23): Reduce the overhead caused by waiting for the Singular prompt by doing garbage collection differently.
- Simon King (2011-06-06): Make conversion from Singular to Sage more flexible.

- Simon King (2015): Extend pickling capabilities.

```
class sage.interfaces.singular.Singular(maxread=None, script_subdirectory=None, logfile=None,
                                         server=None, server_tmpdir=None, seed=None)
```

Bases: sage.interfaces.tab_completion.ExtraTabCompletion, *sage.interfaces.expect.Expect*

Interface to the Singular interpreter.

EXAMPLES: A Groebner basis example.

```
sage: R = singular.ring(0, '(x0,x1,x2)', 'lp')
sage: I = singular.ideal([ 'x0*x1*x2 -x0^2*x2', 'x0^2*x1*x2-x0*x1^2*x2-x0*x1*x2^2',
→ 'x0*x1-x0*x2-x1*x2'])
sage: I.groebner()
x1^2*x2^2,
x0*x2^3-x1^2*x2^2+x1*x2^3,
x0*x1-x0*x2-x1*x2,
x0^2*x2-x0*x2^2-x1*x2^2
```

AUTHORS:

- David Joyner and William Stein

LIB(*lib*, reload=False)

Load the Singular library named lib.

Note that if the library was already loaded during this session it is not reloaded unless the optional reload argument is True (the default is False).

EXAMPLES:

```
sage: singular.lib('sing.lib')
sage: singular.lib('sing.lib', reload=True)
```

clear(*var*)

Clear the variable named var.

EXAMPLES:

```
sage: singular.set('int', 'x', '2')
sage: singular.get('x')
'2'
sage: singular.clear('x')
```

“Clearing the variable” means to allow to free the memory that it uses in the Singular sub-process. However, the actual deletion of the variable is only committed when the next element in the Singular interface is created:

```
sage: singular.get('x')
'2'
sage: a = singular(3)
sage: singular.get('x')
'`x`'
```

console()

EXAMPLES:

```

sage: singular_console() #not tested
             SINGULAR                      / Development
A Computer Algebra System for Polynomial Computations / version 3-0-4
                                                    0<
             by: G.-M. Greuel, G. Pfister, H. Schoenemann \ Nov 2007
FB Mathematik der Universitaet, D-67653 Kaiserslautern \

```

cputime(t=None)

Returns the amount of CPU time that the Singular session has used. If `t` is not None, then it returns the difference between the current CPU time and `t`.

EXAMPLES:

```

sage: t = singular.cputime()
sage: R = singular.ring(0, '(x0,x1,x2)', 'lp')
sage: I = singular.ideal([ 'x0*x1*x2 -x0^2*x2', 'x0^2*x1*x2-x0*x1^2*x2-x0*x1*x2^
↳2', 'x0*x1-x0*x2-x1*x2'])
sage: gb = I.groebner()
sage: singular.cputime(t) #random
0.02

```

current_ring()

Returns the current ring of the running Singular session.

EXAMPLES:

```

sage: r = PolynomialRing(GF(127),3,'xyz', order='invlex')
sage: r._singular_()
polynomial ring, over a field, global ordering
// coefficients: ZZ/127
// number of vars : 3
//      block 1 : ordering rp
//      : names   x y z
//      block 2 : ordering C
sage: singular.current_ring()
polynomial ring, over a field, global ordering
// coefficients: ZZ/127
// number of vars : 3
//      block 1 : ordering rp
//      : names   x y z
//      block 2 : ordering C

```

current_ring_name()

Returns the Singular name of the currently active ring in Singular.

OUTPUT: currently active ring's name

EXAMPLES:

```

sage: r = PolynomialRing(GF(127),3,'xyz')
sage: r._singular_().name() == singular.current_ring_name()
True

```

eval(x, allow_semicolon=True, strip=True, **kws)

Send the code `x` to the Singular interpreter and return the output as a string.

INPUT:

- x - string (of code)
- allow_semicolon - default: False; if False then raise a TypeError if the input line contains a semicolon.
- strip - ignored

EXAMPLES:

```
sage: singular.eval('2 > 1')
'1'
sage: singular.eval('2 + 2')
'4'
```

if the verbosity level is > 1 comments are also printed and not only returned.

```
sage: r = singular.ring(0, '(x,y,z)', 'dp')
sage: i = singular.ideal(['x^2', 'y^2', 'z^2'])
sage: s = i.std()
sage: singular.eval('hilb(%s)%'(s.name()))
'// 1 t^0\n// -3 t^2\n// 3 t^4\n// -1 t^6\n\n// 1 t^0\n//
3 t^1\n// 3 t^2\n// 1 t^3\n// dimension (affine) = 0\n//
degree (affine) = 8'
```

```
sage: from sage.misc.VERBOSE import set_verbosity
sage: set_verbosity(1)
sage: o = singular.eval('hilb(%s)%'(s.name()))
// 1 t^0
// -3 t^2
// 3 t^4
// -1 t^6
// 1 t^0
// 3 t^1
// 3 t^2
// 1 t^3
// dimension (affine) = 0
// degree (affine) = 8
```

This is mainly useful if this method is called implicitly. Because then intermediate results, debugging outputs and printed statements are printed

```
sage: o = s.hilb()
// 1 t^0
// -3 t^2
// 3 t^4
// -1 t^6
// 1 t^0
// 3 t^1
// 3 t^2
// 1 t^3
// dimension (affine) = 0
// degree (affine) = 8
// ** right side is not a datum, assignment ignored
...
```

rather than ignored

```
sage: set_verbose(0)
sage: o = s.hilb()
```

get(*var*)

Get string representation of variable named var.

EXAMPLES:

```
sage: singular.set('int', 'x', '2')
sage: singular.get('x')
'2'
```

ideal(*gens)

Return the ideal generated by gens.

INPUT:

- gens - list or tuple of Singular objects (or objects that can be made into Singular objects via evaluation)

OUTPUT: the Singular ideal generated by the given list of gens

EXAMPLES: A Groebner basis example done in a different way.

```
sage: _ = singular.eval("ring R=0,(x0,x1,x2),lp")
sage: i1 = singular.ideal([ 'x0*x1*x2 -x0^2*x2', 'x0^2*x1*x2-x0*x1^2*x2-
↳ x0*x1*x2^2', 'x0*x1-x0*x2-x1*x2' ])
sage: i1
-x0^2*x2+x0*x1*x2,
x0^2*x1*x2-x0*x1^2*x2-x0*x1*x2^2,
x0*x1-x0*x2-x1*x2
```

```
sage: i2 = singular.ideal('groebner(%s);'%i1.name())
sage: i2
x1^2*x2^2,
x0*x2^3-x1^2*x2^2+x1*x2^3,
x0*x1-x0*x2-x1*x2,
x0^2*x2-x0*x2^2-x1*x2^2
```

lib(*lib*, *reload=False*)

Load the Singular library named lib.

Note that if the library was already loaded during this session it is not reloaded unless the optional reload argument is True (the default is False).

EXAMPLES:

```
sage: singular.lib('sing.lib')
sage: singular.lib('sing.lib', reload=True)
```

list(*x*)

Creates a list in Singular from a Sage list x.

EXAMPLES:

```
sage: singular.list([1,2])
[1]:
  1
```

(continues on next page)

(continued from previous page)

```
[2]:
  2
sage: singular.list([1,2,[3,4]])
[1]:
  1
[2]:
  2
[3]:
  [1]:
    3
  [2]:
    4
sage: R.<x,y> = QQ[]
sage: singular.list([1,2,[x,ideal(x,y)]])
[1]:
  1
[2]:
  2
[3]:
  [1]:
    x
  [2]:
    _[1]=x
    _[2]=y
```

Strings have to be escaped before passing them to this method:

```
sage: singular.list([1,2,'hi'])
[1]:
  1
[2]:
  2
[3]:
  hi
```

load(*lib*, *reload=False*)

Load the Singular library named *lib*.

Note that if the library was already loaded during this session it is not reloaded unless the optional *reload* argument is *True* (the default is *False*).

EXAMPLES:

```
sage: singular.lib('sing.lib')
sage: singular.lib('sing.lib', reload=True)
```

matrix(*nrows*, *ncols*, *entries=None*)

EXAMPLES:

```
sage: singular.lib("matrix")
sage: R = singular.ring(0, '(x,y,z)', 'dp')
sage: A = singular.matrix(3,2,'1,2,3,4,5,6')
```

(continues on next page)

(continued from previous page)

```

sage: A
1, 2,
3, 4,
5, 6
sage: A.gauss_col()
2, -1,
1, 0,
0, 1
    
```

AUTHORS:

- Martin Albrecht (2006-01-14)

option(*cmd=None, val=None*)

Access to Singular's options as follows:

Syntax: option() Returns a string of all defined options.

Syntax: option('option_name') Sets an option. Note to disable an option, use the prefix no.

Syntax: option('get') Returns an intvec of the state of all options.

Syntax: option('set', intvec_expression) Restores the state of all options from an intvec (produced by option('get')).

EXAMPLES:

```

sage: singular.option()
//options: redefine loadLib usage prompt
sage: singular.option('get')
0,
10321
sage: old_options = _
sage: singular.option('noredefine')
sage: singular.option()
//options: loadLib usage prompt
sage: singular.option('set', old_options)
sage: singular.option('get')
0,
10321
    
```

ring(*char=0, vars='(x)', order='lp', check=None*)

Create a Singular ring and makes it the current ring.

INPUT:

- **char** (string) – a string specifying the characteristic of the base ring, in the format accepted by Singular (see examples below).
- **vars** – a tuple or string defining the variable names
- **order** (string) – the monomial order (default: "lp")

OUTPUT: a Singular ring

Note: This function is *not* identical to calling the Singular **ring** function. In particular, it also attempts to “kill” the variable names, so they can actually be used without getting errors, and it sets printing of elements

for this range to short (i.e., with *'s and carets).

EXAMPLES: We first declare $\mathbb{Q}[x, y, z]$ with degree reverse lexicographic ordering.

```

sage: R = singular.ring(0, '(x,y,z)', 'dp')
sage: R
polynomial ring, over a field, global ordering
// coefficients: QQ
// number of vars : 3
//      block   1 : ordering dp
//              : names   x y z
//      block   2 : ordering C

```

```

sage: R1 = singular.ring(32003, '(x,y,z)', 'dp')
sage: R2 = singular.ring(32003, '(a,b,c,d)', 'lp')

```

This is a ring in variables named $x(1)$ through $x(10)$ over the finite field of order 7:

```

sage: R3 = singular.ring(7, '(x(1..10))', 'ds')

```

This is a polynomial ring over the transcendental extension $\mathbb{Q}(a)$ of \mathbb{Q} :

```

sage: R4 = singular.ring('(0,a)', '(mu,nu)', 'lp')

```

This is a ring over the field of single-precision floats:

```

sage: R5 = singular.ring('real', '(a,b)', 'lp')

```

This is over 50-digit floats:

```

sage: R6 = singular.ring('(real,50)', '(a,b)', 'lp')
sage: R7 = singular.ring('(complex,50,i)', '(a,b)', 'lp')

```

To use a ring that you’ve defined, use the `set_ring()` method on the ring. This sets the ring to be the “current ring”. For example,

```

sage: R = singular.ring(7, '(a,b)', 'ds')
sage: S = singular.ring('real', '(a,b)', 'lp')
sage: singular.new('10*a')
(1.000e+01)*a
sage: R.set_ring()
sage: singular.new('10*a')
3*a

```

set(*type, name, value*)

Set the variable with given name to the given value.

REMARK:

If a variable in the Singular interface was previously marked for deletion, the actual deletion is done here, before the new variable is created in Singular.

EXAMPLES:

```

sage: singular.set('int', 'x', '2')
sage: singular.get('x')
'2'
    
```

We test that an unused variable is only actually deleted if this method is called:

```

sage: a = singular(3)
sage: n = a.name()
sage: del a
sage: singular.eval(n)
'3'
sage: singular.set('int', 'y', '5')
sage: singular.eval('defined(%s)%n')
'0'
    
```

set_ring(*R*)

Sets the current Singular ring to *R*.

EXAMPLES:

```

sage: R = singular.ring(7, '(a,b)', 'ds')
sage: S = singular.ring('real', '(a,b)', 'lp')
sage: singular.current_ring()
polynomial ring, over a field, global ordering
// coefficients: Float()
// number of vars : 2
//      block  1 : ordering lp
//              : names  a b
//      block  2 : ordering C
sage: singular.set_ring(R)
sage: singular.current_ring()
polynomial ring, over a field, local ordering
// coefficients: ZZ/7
// number of vars : 2
//      block  1 : ordering ds
//              : names  a b
//      block  2 : ordering C
    
```

set_seed(*seed=None*)

Set the seed for singular interpreter.

The seed should be an integer at least 1 and not more than 30 bits. See http://www.singular.uni-kl.de/Manual/html/sing_19.htm#SEC26 and http://www.singular.uni-kl.de/Manual/html/sing_283.htm#SEC323

EXAMPLES:

```

sage: s = Singular()
sage: s.set_seed(1)
1
sage: [s.random(1,10) for i in range(5)]
[8, 10, 4, 9, 1]
    
```

setring(*R*)

Sets the current Singular ring to *R*.

EXAMPLES:

```
sage: R = singular.ring(7, '(a,b)', 'ds')
sage: S = singular.ring('real', '(a,b)', 'lp')
sage: singular.current_ring()
polynomial ring, over a field, global ordering
//  coefficients: Float()
//  number of vars : 2
//      block  1 : ordering lp
//           : names  a b
//      block  2 : ordering C
sage: singular.set_ring(R)
sage: singular.current_ring()
polynomial ring, over a field, local ordering
//  coefficients: ZZ/7
//  number of vars : 2
//      block  1 : ordering ds
//           : names  a b
//      block  2 : ordering C
```

string(*x*)

Creates a Singular string from a Sage string. Note that the Sage string has to be “double-quoted”.

EXAMPLES:

```
sage: singular.string('"Sage"')
Sage
```

version()

Return the version of Singular being used.

EXAMPLES:

```
sage: singular.version()
"Singular ... version 4..."
```

class sage.interfaces.singular.SingularElement(*parent, type, value, is_name=False*)

Bases: `sage.interfaces.tab_completion.ExtraTabCompletion`, `sage.interfaces.expect.ExpectElement`

EXAMPLES:

```
sage: a = singular(2)
sage: loads(dumps(a))
2
```

attrib(*name, value=None*)

Get and set attributes for self.

INPUT:

- *name* - string to choose the attribute
- *value* - boolean value or None for reading, (default:None)

VALUES: *isSB* - the standard basis property is set by all commands computing a standard basis like groebner, std, stdhilb etc.; used by lift, dim, degree, mult, hilb, vdim, kbase *isHomog* - the weight vector for

homogeneous or quasihomogeneous ideals/modules isCI - complete intersection property isCM - Cohen-Macaulay property rank - set the rank of a module (see nrows) withSB - value of type ideal, resp. module, is std withHilb - value of type intvec is hilb(.,1) (see hilb) withRes - value of type list is a free resolution withDim - value of type int is the dimension (see dim) withMult - value of type int is the multiplicity (see mult)

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([z^2, y*z, y^2, x*z, x*y, x^2])
sage: Ibar = I._singular_()
sage: Ibar.attrib('isSB')
0
sage: singular.eval('vdim(%s)%Ibar.name()') # sage7 name is random
// ** sage7 is no standard basis
4
sage: Ibar.attrib('isSB',1)
sage: singular.eval('vdim(%s)%Ibar.name()')
'4'
```

is_string()

Tell whether this element is a string.

EXAMPLES:

```
sage: singular('"abc").is_string()
True
sage: singular('1').is_string()
False
```

sage_flattened_str_list()

EXAMPLES:

```
sage: R=singular.ring(0,'(x,y)', 'dp')
sage: RL = R.ringlist()
sage: RL.sage_flattened_str_list()
['0', 'x', 'y', 'dp', '1,1', 'C', '0', '_[1]=0']
```

sage_global_ring()

Return the current basering in Singular as a polynomial ring or quotient ring.

EXAMPLES:

```
sage: singular.eval('ring r1 = (9,x),(a,b,c,d,e,f),(M((1,2,3,0)),wp(2,3),lp)')
''
sage: R = singular('r1').sage_global_ring()
sage: R
Multivariate Polynomial Ring in a, b, c, d, e, f over Finite Field in x of size_
↪ 3^2
sage: R.term_order()
Block term order with blocks:
(Matrix term order with matrix
[1 2]
[3 0]),
Weighted degree reverse lexicographic term order with weights (2, 3),
Lexicographic term order of length 2)
```

```
sage: singular.eval('ring r2 = (0,x),(a,b,c),dp')
''
sage: singular('r2').sage_global_ring()
Multivariate Polynomial Ring in a, b, c over Fraction Field of Univariate
↳ Polynomial Ring in x over Rational Field
```

```
sage: singular.eval('ring r3 = (3,z),(a,b,c),dp')
''
sage: singular.eval('minpoly = 1+z+z2+z3+z4')
''
sage: singular('r3').sage_global_ring()
Multivariate Polynomial Ring in a, b, c over Finite Field in z of size 3^4
```

Real and complex fields in both Singular and Sage are defined with a precision. The precision in Singular is given in terms of digits, but in Sage it is given in terms of bits. So, the digit precision is internally converted to a reasonable bit precision:

```
sage: singular.eval('ring r4 = (real,20),(a,b,c),dp')
''
sage: singular('r4').sage_global_ring()
Multivariate Polynomial Ring in a, b, c over Real Field with 70 bits of
↳ precision
```

The case of complex coefficients is not fully supported, yet, since the generator of a complex field in Sage is always called “I”:

```
sage: singular.eval('ring r5 = (complex,15,j),(a,b,c),dp')
''
sage: R = singular('r5').sage_global_ring(); R
Multivariate Polynomial Ring in a, b, c over Complex Field with 54 bits of
↳ precision
sage: R.base_ring()['k']
Traceback (most recent call last):
...
ValueError: given string 'k' is not a complex number
sage: R.base_ring()['I']
1.0000000000000000*I
```

An example where the base ring is a polynomial ring over an extension of the rational field:

```
sage: singular.eval('ring r7 = (0,a), (x,y), dp')
''
sage: singular.eval('minpoly = a2 + 1')
''
sage: singular('r7').sage_global_ring()
Multivariate Polynomial Ring in x, y over Number Field in a with defining
↳ polynomial a^2 + 1
```

In our last example, the base ring is a quotient ring:

```
sage: singular.eval('ring r6 = (9,a), (x,y,z),lp')
''
sage: Q = singular('std(ideal(x^2,x+y^2+z^3))', type='qring')
```

(continues on next page)

(continued from previous page)

```
sage: Q.sage_global_ring()
Quotient of Multivariate Polynomial Ring in x, y, z over Finite Field in a of
↳size 3^2 by the ideal (y^4 - y^2*z^3 + z^6, x + y^2 + z^3)
```

AUTHOR:

- Simon King (2011-06-06)

sage_matrix(*R, sparse=True*)

Returns Sage matrix for self

INPUT:

- *R* - (default: None); an optional ring, over which the resulting matrix is going to be defined. By default, the output of *sage_global_ring()* is used.
- *sparse* - (default: True); determines whether the resulting matrix is sparse or not.

EXAMPLES:

```
sage: R = singular.ring(0, '(x,y,z)', 'dp')
sage: A = singular.matrix(2,2)
sage: A.sage_matrix(ZZ)
[0 0]
[0 0]
sage: A.sage_matrix(RDF)
[0.0 0.0]
[0.0 0.0]
```

sage_poly(*R=None, kcache=None*)

Returns a Sage polynomial in the ring *r* matching the provided poly which is a singular polynomial.

INPUT:

- *R* - (default: None); an optional polynomial ring. If it is provided, then you have to make sure that it matches the current singular ring as, e.g., returned by *singular.current_ring()*. By default, the output of *sage_global_ring()* is used.
- *kcache* - (default: None); an optional dictionary for faster finite field lookups, this is mainly useful for finite extension fields

OUTPUT: MPolynomial

EXAMPLES:

```
sage: R = PolynomialRing(GF(2^8, 'a'), 'x,y')
sage: f = R('a^20*x^2*y+a^10+x')
sage: f._singular().sage_poly(R) == f
True
sage: R = PolynomialRing(GF(2^8, 'a'), 'x', implementation="singular")
sage: f = R('a^20*x^3+x^2+a^10')
sage: f._singular().sage_poly(R) == f
True
```

```
sage: P.<x,y> = PolynomialRing(QQ, 2)
sage: f = x*y**3 - 1/9 * x + 1; f
x*y^3 - 1/9*x + 1
```

(continues on next page)

(continued from previous page)

```

sage: singular(f)
x*y^3-1/9*x+1
sage: P(singular(f))
x*y^3 - 1/9*x + 1
  
```

AUTHORS:

- Martin Albrecht (2006-05-18)
- Simon King (2011-06-06): Deal with Singular's short polynomial representation, automatic construction of a polynomial ring, if it is not explicitly given.

Note: For very simple polynomials `eval(SingularElement.sage_polystring())` is faster than `SingularElement.sage_poly(R)`, maybe we should detect the crossover point (in dependence of the string length) and choose an appropriate conversion strategy

sage_polystring()

If this Singular element is a polynomial, return a string representation of this polynomial that is suitable for evaluation in Python. Thus `*` is used for multiplication and `**` for exponentiation. This function is primarily used internally.

The `short=0` option *must* be set for the parent ring or this function will not work as expected. This option is set by default for rings created using `singular.ring` or set using `ring_name.set_ring()`.

EXAMPLES:

```

sage: R = singular.ring(0, '(x,y)')
sage: f = singular('x^3 + 3*y^11 + 5')
sage: f
x^3+3*y^11+5
sage: f.sage_polystring()
'x**3+3*y**11+5'
  
```

sage_structured_str_list()

If `self` is a Singular list of lists of Singular elements, returns corresponding Sage list of lists of strings.

EXAMPLES:

```

sage: R=singular.ring(0, '(x,y)', 'dp')
sage: RL=R.ringlist()
sage: RL
[1]:
  0
[2]:
  [1]:
    x
  [2]:
    y
[3]:
  [1]:
    [1]:
      dp
    [2]:
      1,1
  
```

(continues on next page)

(continued from previous page)

```

[2]:
  [1]:
    C
  [2]:
    0
[4]:
  _[1]=0
sage: RL.sage_structured_str_list()
['0', ['x', 'y'], [['dp', '1,\n1'], ['C', '0']], '0']
    
```

set_ring()

Sets the current ring in Singular to be self.

EXAMPLES:

```

sage: R = singular.ring(7, '(a,b)', 'ds')
sage: S = singular.ring('real', '(a,b)', 'lp')
sage: singular.current_ring()
polynomial ring, over a field, global ordering
// coefficients: Float()
// number of vars : 2
//      block  1 : ordering lp
//              : names   a b
//      block  2 : ordering C
sage: R.set_ring()
sage: singular.current_ring()
polynomial ring, over a field, local ordering
// coefficients: ZZ/7
// number of vars : 2
//      block  1 : ordering ds
//              : names   a b
//      block  2 : ordering C
    
```

type()

Returns the internal type of this element.

EXAMPLES:

```

sage: R = PolynomialRing(GF(2^8, 'a'), 2, 'x')
sage: R._singular().type()
'ring'
sage: fs = singular('x0^2', 'poly')
sage: fs.type()
'poly'
    
```

exception `sage.interfaces.singular.SingularError`

Bases: `RuntimeError`

Raised if Singular printed an error message

class `sage.interfaces.singular.SingularFunction`(*parent, name*)

Bases: `sage.interfaces.expect.ExpectFunction`

class `sage.interfaces.singular.SingularFunctionElement`(*obj, name*)

Bases: `sage.interfaces.expect.FunctionElement`

`class sage.interfaces.singular.SingularGBDefaultContext(singular=None)`

Bases: object

Within this context all Singular Groebner basis calculations are reduced automatically.

AUTHORS:

- Martin Albrecht
- Simon King

`class sage.interfaces.singular.SingularGBLogPrettyPrinter(verbosity=1)`

Bases: object

A device which prints Singular Groebner basis computation logs more verbatim.

`flush()`

EXAMPLES:

```
sage: from sage.interfaces.singular import SingularGBLogPrettyPrinter
sage: s3 = SingularGBLogPrettyPrinter(verbosity=3)
sage: s3.flush()
```

`write(s)`

EXAMPLES:

```
sage: from sage.interfaces.singular import SingularGBLogPrettyPrinter
sage: s3 = SingularGBLogPrettyPrinter(verbosity=3)
sage: s3.write("S:1337")
Performing complete reduction of 1337 elements.
sage: s3.write("M[389,12]")
Parallel reduction of 389 elements with 12 non-zero output elements.
```

`sage.interfaces.singular.generate_docstring_dictionary()`

Generate global dictionaries which hold the docstrings for Singular functions.

EXAMPLES:

```
sage: from sage.interfaces.singular import generate_docstring_dictionary
sage: generate_docstring_dictionary()
```

`sage.interfaces.singular.get_docstring(name)`

Return the docstring for the function name.

INPUT:

- name - a Singular function name

EXAMPLES:

```
sage: from sage.interfaces.singular import get_docstring
sage: 'groebner' in get_docstring('groebner')
True
sage: 'standard.lib' in get_docstring('groebner')
True
```

`sage.interfaces.singular.is_SingularElement(x)`

Returns True if x is of type SingularElement.

EXAMPLES:

```
sage: from sage.interfaces.singular import is_SingularElement
sage: is_SingularElement(singular(2))
True
sage: is_SingularElement(2)
False
```

sage.interfaces.singular.reduce_load_Singular()

EXAMPLES:

```
sage: from sage.interfaces.singular import reduce_load_Singular
sage: reduce_load_Singular()
Singular
```

sage.interfaces.singular.singular_console()

Spawn a new Singular command-line session.

EXAMPLES:

```
sage: singular_console() #not tested
                SINGULAR / Development
A Computer Algebra System for Polynomial Computations / version 3-0-4
                0<
      by: G.-M. Greuel, G. Pfister, H. Schoenemann \ Nov 2007
FB Mathematik der Universitaet, D-67653 Kaiserslautern \
```

sage.interfaces.singular.singular_gb_standard_options(*func*)

Decorator to force a reduced Singular groebner basis.

Note: This decorator is used automatically internally so the user does not need to use it manually.

sage.interfaces.singular.singular_version()

Return the version of Singular being used.

EXAMPLES:

```
sage: singular.version()
"Singular ... version 4..."
```


SYMPY → SAGE CONVERSION

The file consists of `_sage_()` methods that are added lazily to the respective SymPy objects. Any call of the `_sympy_()` method of a symbolic expression will trigger the addition. See `sage.symbolic.expression_conversion.SymPyConverter` for the conversion to SymPy.

Only Function objects where the names differ need their own `_sage_()` method. There are several functions with differing name that have an alias in Sage that is the same as the name in SymPy, so no explicit translation is needed for them:

```
sage: from sympy import Symbol, Si, Ci, Shi, Chi, sign
sage: sx = Symbol('x')
sage: assert sin_integral(x)._sympy_() == Si(sx)
sage: assert sin_integral(x) == Si(sx)._sage_()
sage: assert sinh_integral(x)._sympy_() == Shi(sx)
sage: assert sinh_integral(x) == Shi(sx)._sage_()
sage: assert cos_integral(x)._sympy_() == Ci(sx)
sage: assert cos_integral(x) == Ci(sx)._sage_()
sage: assert cosh_integral(x)._sympy_() == Chi(sx)
sage: assert cosh_integral(x) == Chi(sx)._sage_()
sage: assert sgn(x)._sympy_() == sign(sx)
sage: assert sgn(x) == sign(sx)._sage_()
```

AUTHORS:

- Ralf Stephan (2017-10)

class `sage.interfaces.sympy.UndefSageHelper`

Bases: `object`

Helper class to convert sympy function objects to sage functions

EXAMPLES:

```
sage: from sympy import Function
sage: f = function('f')
sage: F = Function('f')
sage: assert f._sympy_() == F
sage: assert f == F._sage_()
```

`sage.interfaces.sympy.check_expression(expr, var_symbols, only_from_sympy=False)`

Does `eval(expr)` both in Sage and SymPy and does other checks.

EXAMPLES:

```
sage: from sage.interfaces.sympy import check_expression
sage: check_expression("1.123*x", "x")
```

sage.interfaces.sympy.**sympy_init**(**args*, ***kwargs*)

Add `_sage_()` methods to SymPy objects where needed.

This gets called with every call to `Expression._sympy_()` so there is only need to call it if you bypass `_sympy_()` to create SymPy objects. Note that SymPy objects have `_sage_()` methods hard installed but having them inside Sage as one file makes them easier to maintain for Sage developers.

EXAMPLES:

```
sage: from sage.interfaces.sympy import sympy_init
sage: from sympy import Symbol, Abs
sage: sympy_init()
sage: assert abs(x) == Abs(Symbol('x'))._sage_()
```

sage.interfaces.sympy.**sympy_set_to_list**(*set*, *vars*)

Convert all set objects that can be returned by SymPy's solvers.

sage.interfaces.sympy.**test_all**()

Call some tests that were originally in SymPy.

EXAMPLES:

```
sage: from sage.interfaces.sympy import test_all
sage: test_all()
```

THE TACHYON RAY TRACER

AUTHOR:

- John E. Stone

This documentation, which was written by John Stone, describes how to create scene files.

At the present time, scene description files are very simple. The parser can't handle multiple file scene descriptions, although they may be added in the future. Most of the objects and their scene description are closely related to the RAY API. (*See the API docs for additional info.*)

46.1 Basic Scene Requirements

Unlike some other ray tracers out there, RAY requires that you specify most of the scene parameters in the scene description file itself. If users would rather specify some of these parameters at the command line, then I may add that feature in the future. A scene description file contains keywords, and values associated or grouped with a keyword. All keywords can be in caps, lower case, or mixed case for the convenience of the user. File names and texture names are normally case-sensitive, although the behavior for file names is operating system-dependent. All values are either character strings, or floating point numbers. In some cases, the presence of one keyword will require additional keyword / value pairs.

At the moment there are several keywords with values, that must appear in every scene description file. Every scene description file must begin with the `BEGIN_SCENE` keyword, and end with the `END_SCENE` keyword. All definitions and declarations of any kind must be inside the `BEGIN_SCENE`, `END_SCENE` pair. The `RESOLUTION` keyword is followed by an x resolution and a y resolution in terms of pixels on each axis. There are currently no limits placed on the resolution of an output image other than the computer's available memory and reasonable execution time. An example of a simple scene description skeleton is shown below:

```
BEGIN_SCENE
  RESOLUTION 1024 1024
...
...  Camera definition..
...
...  Other objects, etc..
...
END_SCENE
```

46.2 Camera and viewing parameters

One of the most important parts of any scene, is the camera position and orientation. Having a good angle on a scene can make the difference between an average looking scene and a strikingly interesting one. There may be multiple camera definitions in a scene file, but the last camera definition overrides all previous definitions. There are several parameters that control the camera in , PROJECTION, ZOOM, ASPECTRATIO, ANTIALIASING, CENTER, RAYDEPTH, VIEWDIR, and UPDIR.

The first and last keywords required in the definition of a camera are the CAMERA and END_CAMERA keywords. The PROJECTION keyword is optional, the remaining camera keywords are required, and must be written in the sequence they are listed in the examples in this section.

46.2.1 Camera projection modes

The PROJECTION keyword must be followed by one of the supported camera projection mode identifiers PERSPECTIVE, PERSPECTIVE_DOF, ORTHOGRAPHIC, or FISHEYE. The FISHEYE projection mode requires two extra parameters FOCALLENGTH and APERTURE which precede the regular camera options.

```

Camera
  projection perspective_dof
  focallength 0.75
  aperture 0.02
  Zoom 0.666667
  Aspectratio 1.000000
  Antialiasing 128
  Raydepth 30
  Center 0.000000 0.000000 -2.000000
  Viewdir -0.000000 -0.000000 2.000000
  Updir 0.000000 1.000000 -0.000000
End_Camera

```

46.2.2 Common camera parameters

The ZOOM parameter controls the camera in a way similar to a telephoto lens on a 35mm camera. A zoom value of 1.0 is standard, with a 90 degree field of view. By changing the zoom factor to 2.0, the relative size of any feature in the frame is twice as big, while the field of view is decreased slightly. The zoom effect is implemented as a scaling factor on the height and width of the image plane relative to the world.

The ASPECTRATIO parameter controls the aspect ratio of the resulting image. By using the aspect ratio parameter, one can produce images which look correct on any screen. Aspect ratio alters the relative width of the image plane, while keeping the height of the image plane constant. In general, most workstation displays have an aspect ratio of 1.0. To see what aspect ratio your display has, you can render a simple sphere, at a resolution of 512x512 and measure the ratio of its width to its height.

The ANTIALIASING parameter controls the maximum level of supersampling used to obtain higher image quality. The parameter given sets the number of additional rays to trace per-pixel to attain higher image quality.

The RAYDEPTH parameter tells RAY what the maximum level of reflections, refraction, or in general the maximum recursion depth to trace rays to. A value between 4 and 12 is usually good. A value of 1 will disable rendering of reflective or transmissive objects (they'll be black).

The remaining three camera parameters are the most important, because they define the coordinate system of the camera, and its position in the scene. The CENTER parameter is an X, Y, Z coordinate defining the center of the camera (*also known as the Center of Projection*). Once you have determined where the camera will be placed in the scene, you need

to tell RAY what the camera should be looking at. The VIEWDIR parameter is a vector indicating the direction the camera is facing. It may be useful for me to add a “Look At” type keyword in the future to make camera aiming easier. If people want or need the “Look At” style camera, let me know. The last parameter needed to completely define a camera is the “up” direction. The UPDIR parameter is a vector which points in the direction of the “sky”. I wrote the camera so that VIEWDIR and UPDIR don’t have to be perpendicular, and there shouldn’t be a need for a “right” vector although some other ray tracers require it. Here’s a snippet of a camera definition:

```
CAMERA
  ZOOM 1.0
  ASPECTRATIO 1.0
  ANTIALIASING 0
  RAYDEPTH 12
  CENTER 0.0 0.0 2.0
  VIEWDIR 0 0 -1
  UPDIR 0 1 0
END_CAMERA
```

46.2.3 Viewing frustum

An optional FRUSTUM parameter provides a means for rendering sub-images in a larger frame, and correct stereoscopic images. The FRUSTUM keyword must be followed by four floating parameters, which indicate the top, bottom, left and right coordinates of the image plane in eye coordinates. When the projection mode is set to FISHEYE, the frustum parameters correspond to spherical coordinates specified in radians.

```
CAMERA
  ZOOM 1.0
  ASPECTRATIO 1.0
  ANTIALIASING 0
  RAYDEPTH 4
  CENTER 0.0 0.0 -6.0
  VIEWDIR 0.0 0.0 1.0
  UPDIR 0.0 1.0 0.0
  FRUSTUM -0.5 0.5 -0.5 0.5
END_CAMERA
```

46.3 Including Files

The INCLUDE keyword is used anywhere after the camera description, and is immediately followed by a valid filename, for a file containing additional scene description information. The included file is opened, and processing continues as if it were part of the current file, until the end of the included file is reached. Parsing of the current file continues from where it left off prior to the included file.

46.4 Scene File Comments

The # keyword is used anywhere after the camera description, and will cause RAY to ignore all characters from the # to the end of the input line. The # character must be surrounded by whitespace in order to be recognized. A sequence such as ### will not be recognized as a comment.

46.5 Lights

The most frequently used type of lights provided by RAY are positional point light sources. The lights are actually small spheres, which are visible. A point light is composed of three pieces of information, a center, a radius (since its a sphere), and a color. To define a light, simply write the LIGHT keyword, followed by its CENTER (a X, Y, Z coordinate), its RAD (radius, a scalar), and its COLOR (a Red Green Blue triple). The radius parameter will accept any value of 0.0 or greater. Lights of radius 0.0 will not be directly visible in the rendered scene, but contribute light to the scene normally. For a light, the color values range from 0.0 to 1.0, any values outside this range may yield unpredictable results. A simple light definition looks like this:

```
LIGHT CENTER 4.0 3.0 2.0
      RAD   0.2
      COLOR 0.5 0.5 0.5
```

This light would be gray colored if seen directly, and would be 50% intensity in each RGB color component.

RAY supports simple directional lighting, commonly used in CAD and scientific visualization programs for its performance advantages over positional lights. Directional lights cannot be seen directly in scenes rendered by , only their illumination contributes to the final image.

```
DIRECTIONAL_LIGHT
  DIRECTION 0.0 -1.0 0.0
  COLOR    1.0 0.0 0.0
```

RAY supports spotlights, which are described very similarly to a point light, but they are attenuated by angle from the direction vector, based on a “falloff start” angle and “falloff end” angle. Between the starting and ending angles, the illumination is attenuated linearly. The syntax for a spotlight description in a scene file is as follows.

```
SPOTLIGHT
  CENTER 0.0 3.0 17.0
  RAD    0.2
  DIRECTION 0.0 -1.0 0.0
  FALLOFF_START 20.0
  FALLOFF_END 45.0
  COLOR 1.0 0.0 0.0
```

The lighting system implemented by RAY provides various levels of distance-based lighting attenuation. By default, a light is not attenuated by distance. If the *attenuation* keywords is present immediately prior to the light’s color, RAY will accept coefficients which are used to calculate distance-based attenuation, which is applied the light by multiplying with the resulting value. The attenuation factor is calculated from the equation

$$1/(K_c + K_l d + k_q d^2)$$

This attenuation equation should be familiar to some as it is the same lighting attenuation equation used by OpenGL. The constant, linear, and quadratic terms are specified in a scene file as shown in the following example.

```

LIGHT
  CENTER -5.0 0.0 10.0
  RAD 1.0
  ATTENUATION CONSTANT 1.0 LINEAR 0.2 QUADRATIC 0.05
  COLOR 1.0 0.0 0.0
    
```

46.6 Atmospheric effects

RAY currently only implements one atmospheric effect, simple distance-based fog.

46.6.1 Fog

RAY provides a simple distance-based fog effect intended to provide functionality similar to that found in OpenGL, for compatibility with software that requires an OpenGL-like fog implementation. Much like OpenGL, RAY provides linear, exponential, and exponential-squared fog.

```

FOG
  LINEAR START 0.0 END 50.0 DENSITY 1.0 COLOR 1.0 1.0 1.0
    
```

```

FOG
  EXP START 0.0 END 50.0 DENSITY 1.0 COLOR 1.0 1.0 1.0
    
```

```

FOG
  EXP2 START 0.0 END 50.0 DENSITY 1.0 COLOR 1.0 1.0 1.0
    
```

46.7 Objects

46.7.1 Spheres

Spheres are the simplest object supported by RAY and they are also the fastest object to render. Spheres are defined as one would expect, with a CENTER, RAD (radius), and a texture. The texture may be defined along with the object as discussed earlier, or it may be declared and assigned a name. Here's a sphere definition using a previously defined "NitrogenAtom" texture:

```

SPHERE CENTER 26.4 27.4 -2.4 RAD 1.0 NitrogenAtom
    
```

A sphere with an inline texture definition is declared like this:

```

Sphere center 1.0 0.0 10.0
  Rad 1.0
  Texture Ambient 0.2 Diffuse 0.8 Specular 0.0 Opacity 1.0
    Color 1.0 0.0 0.5
    TexFunc 0
    
```

Notice that in this example I used mixed case for the keywords, this is allowable... Review the section on textures if the texture definitions are confusing.

46.7.2 Triangles

Triangles are also fairly simple objects, constructed by listing the three vertices of the triangle, and its texture. The order of the vertices isn't important, the triangle object is "double sided", so the surface normal is always pointing back in the direction of the incident ray. The triangle vertices are listed as V1, V2, and V3 each one is an X, Y, Z coordinate. An example of a triangle is shown below:

```

TRI
  V0  0.0 -4.0 12.0
  V1  4.0 -4.0 8.0
  V2 -4.0 -4.0 8.0
TEXTURE
  AMBIENT 0.1 DIFFUSE  0.2 SPECULAR 0.7 OPACITY 1.0
  COLOR 1.0 1.0 1.0
  TEXTFUNC 0
    
```

46.7.3 Smoothed Triangles

Smoothed triangles are just like regular triangles, except that the surface normal for each of the three vertices is used to determine the surface normal across the triangle by linear interpolation. Smoothed triangles yield curved looking objects and have nice reflections.

```

STRI
  V0 1.4  0.0  2.4
  V1 1.35 -0.37 2.4
  V2 1.36 -0.32 2.45
  N0 -0.9 -0.0 -0.4
  N1 -0.8  0.23 -0.4
  N2 -0.9  0.27 -0.15
TEXTURE
  AMBIENT 0.1 DIFFUSE  0.2 SPECULAR 0.7 OPACITY 1.0
  COLOR 1.0 1.0 1.0
  TEXTFUNC 0
    
```

46.7.4 Infinite Planes

Useful for things like desert floors, backgrounds, skies etc, the infinite plane is pretty easy to use. An infinite plane only consists of two pieces of information, the CENTER of the plane, and a NORMAL to the plane. The center of the plane is just any point on the plane such that the point combined with the surface normal define the equation for the plane. As with triangles, planes are double sided. Here is an example of an infinite plane:

```

PLANE
  CENTER 0.0 -5.0 0.0
  NORMAL 0.0  1.0 0.0
TEXTURE
  AMBIENT 0.1 DIFFUSE  0.9 SPECULAR 0.0  OPACITY 1.0
  COLOR  1.0 1.0 1.0
  TEXTFUNC 1
  CENTER 0.0 -5.0 0.0
  ROTATE 0. 0.0 0.0
  SCALE  1.0 1.0 1.0
    
```

46.7.5 Rings

Rings are a simple object, they are really a not-so-infinite plane. Rings are simply an infinite plane cut into a washer shaped ring, infinitely thing just like a plane. A ring only requires two more pieces of information than an infinite plane does, an inner and outer radius. Here's an example of a ring:

```
Ring  
Center 1.0 1.0 1.0  
Normal 0.0 1.0 0.0  
Inner 1.0  
Outer 5.0  
MyNewRedTexture
```

46.7.6 Infinite Cylinders

Infinite cylinders are quite simple. They are defined by a center, an axis, and a radius. An example of an infinite cylinder is:

```
Cylinder  
Center 0.0 0.0 0.0  
Axis 0.0 1.0 0.0  
Rad 1.0  
SomeRandomTexture
```

46.7.7 Finite Cylinders

Finite cylinders are almost the same as infinite ones, but the center and length of the axis determine the extents of the cylinder. The finite cylinder is also really a shell, it doesn't have any caps. If you need to close off the ends of the cylinder, use two ring objects, with the inner radius set to 0.0 and the normal set to be the axis of the cylinder. Finite cylinders are built this way to enhance speed.

```
FCylinder  
Center 0.0 0.0 0.0  
Axis 0.0 9.0 0.0  
Rad 1.0  
SomeRandomTexture
```

This defines a finite cylinder with radius 1.0, going from 0.0 0.0 0.0, to 0.0 9.0 0.0 along the Y axis. The main difference between an infinite cylinder and a finite cylinder is in the interpretation of the `AXIS` parameter. In the case of the infinite cylinder, the length of the axis vector is ignored. In the case of the finite cylinder, the axis parameter is used to determine the length of the overall cylinder.

46.7.8 Axis Aligned Boxes

Axis aligned boxes are fast, but of limited usefulness. As such, I'm not going to waste much time explaining 'em. An axis aligned box is defined by a **MIN** point, and a **MAX** point. The volume between the min and max points is the box. Here's a simple box:

```
BOX
  MIN -1.0 -1.0 -1.0
  MAX  1.0  1.0  1.0
  Boxtexture1
```

46.7.9 Fractal Landscapes

Currently fractal landscapes are a built-in function. In the near future I'll allow the user to load an image map for use as a heightfield. Fractal landscapes are currently forced to be axis aligned. Any suggestion on how to make them more appealing to users is welcome. A fractal landscape is defined by its "resolution" which is the number of grid points along each axis, and by its scale and center. The "scale" is how large the landscape is along the X, and Y axes in world coordinates. Here's a simple landscape:

```
SCAPE
  RES 30 30
  SCALE 80.0 80.0
  CENTER 0.0 -4.0 20.0
  TEXTURE
    AMBIENT 0.1 DIFFUSE 0.9 SPECULAR 0.0 OPACITY 1.0
    COLOR 1.0 1.0 1.0
    TEXFUNC 0
```

The landscape shown above generates a square landscape made of 1,800 triangles. When time permits, the heightfield code will be rewritten to be more general and to increase rendering speed.

46.7.10 Arbitrary Quadric Surfaces

Docs soon. I need to add these into the parser, must have forgotten before ;-)

46.7.11 Volume Rendered Scalar Voxels

These are a little trickier than the average object :-). These are likely to change substantially in the very near future so I'm not going to get too detailed yet. A volume rendered data set is described by its axis aligned bounding box, and its resolution along each axis. The final parameter is the voxel data file. If you are seriously interested in messing with these, get hold of me and I'll give you more info. Here's a quick example:

```
SCALARVOL
  MIN -1.0 -1.0 -0.4
  MAX  1.0  1.0  0.4
  DIM 256 256 100
  FILE /cfs/johns/vol/engine.256x256x110
  TEXTURE
    AMBIENT 1.0 DIFFUSE 0.0 SPECULAR 0.0 OPACITY 8.1
    COLOR 1.0 1.0 1.0
    TEXFUNC 0
```


- **outfile** - (default: 'sage.png') output filename; the extension of the filename determines the type. Supported types include:
 - tga - 24-bit (uncompressed)
 - bmp - 24-bit Windows BMP (uncompressed)
 - ppm - 24-bit PPM (uncompressed)
 - rgb - 24-bit SGI RGB (uncompressed)
 - png - 24-bit PNG (compressed, lossless)
- **verbose** - integer; (default: 1)
 - 0 - silent
 - 1 - some output
 - 2 - very verbose output
- **block** - bool (default: True); if False, run the rendering command in the background.
- **extra_opts** - passed directly to tachyon command line. Use `tachyon_rt.usage()` to see some of the possibilities.

OUTPUT:

- Some text may be displayed onscreen.
- The file `outfile` is created.

EXAMPLES:

```
__call__(model, outfile='sage.png', verbose=1, extra_opts='')
This executes the tachyon program, given a scene file input.
```

INPUT:

- **model** – string. The tachyon model.
- **outfile** – string, default 'sage.png'. The filename to save the model to.
- **verbose** – 0, 1, (default) or 2. The verbosity level.
- **extra_opts** – string (default: empty string). Extra options that will be appended to the tachyon commandline.

EXAMPLES:

```
sage: from sage.interfaces.tachyon import TachyonRT
sage: tgen = Tachyon()
sage: tgen.texture('t1')
sage: tgen.sphere((0,0,0),1,'t1')
sage: tgen.str()[30:40]
'resolution'
sage: t = TachyonRT()
sage: import os
sage: t(tgen.str(), outfile=os.devnull)
tachyon ...
Tachyon Parallel/Multiprocessor Ray Tracer...
```

help(*use_pager*=True)

Deprecated: type 'sage.interfaces.tachyon?' for help

usage(*use_pager=True*)

Returns the basic description of using the Tachyon raytracer (simply what is returned by running tachyon with no input). The output is paged unless *use_pager=False*.

INTERFACE TO TIDES

This module contains tools to write the .c files needed for TIDES [TIDES] .

Tides is an integration engine based on the Taylor method. It is implemented as a c library. The user must translate its initial value problem (IVP) into a pair of .c files that will then be compiled and linked against the TIDES library. The resulting binary will produce the desired output. The tools in this module can be used to automate the generation of these files from the symbolic expression of the differential equation.

```
#####  
# Copyright (C) 2014 Miguel Marco <mmarco@unizar.es>, Marcos Rodriguez  
# <marcos@uunizar.es>  
#  
# Distributed under the terms of the GNU General Public License (GPL):  
#  
# http://www.gnu.org/licenses/  
#####
```

AUTHORS:

- Miguel Marco (06-2014) - Implementation of tides solver
- Marcos Rodriguez (06-2014) - Implementation of tides solver
- Alberto Abad (06-2014) - tides solver
- Roberto Barrio (06-2014) - tides solver

REFERENCES:

- [ABBR2012]
- [TIDES]

`sage.interfaces.tides.genfiles_mintides`(*integrator, driver, f, ics, initial, final, delta, tolrel=1e-16, tolabs=1e-16, output=""*)

Generate the needed files for the min_tides library.

INPUT:

- `integrator` – the name of the integrator file.
- `driver` – the name of the driver file.
- `f` – the function that determines the differential equation.
- `ics` – a list or tuple with the initial conditions.
- `initial` – the initial time for the integration.
- `final` – the final time for the integration.

- `delta` – the step of the output.
- `tolrel` – the relative tolerance.
- `tolabs` – the absolute tolerance.
- `output` – the name of the file that the compiled integrator will write to

This function creates two files, `integrator` and `driver`, that can be used later with the `min_tides` library [TIDES].

```
sage.interfaces.tides.genfiles_mpfr(integrator, driver, f, ics, initial, final, delta, parameters=None,
                                   parameter_values=None, dig=20, tolrel=1e-16, tolabs=1e-16,
                                   output="")
```

Generate the needed files for the `mpfr` module of the `tides` library.

INPUT:

- `integrator` – the name of the integrator file.
- `driver` – the name of the driver file.
- `f` – the function that determines the differential equation.
- `ics` – a list or tuple with the initial conditions.
- `initial` – the initial time for the integration.
- `final` – the final time for the integration.
- `delta` – the step of the output.
- **parameters** – the variables inside the function that should be treated as parameters.
- **parameter_values** – the values of the parameters for the particular initial value problem.
- `dig` – the number of digits of precision that will be used in the integration
- `tolrel` – the relative tolerance.
- `tolabs` – the absolute tolerance.
- `output` – the name of the file that the compiled integrator will write to

This function creates two files, `integrator` and `driver`, that can be used later with the `tides` library ([TIDES]).

```
sage.interfaces.tides.remove_constants(l1, l2)
```

Given two lists, remove the entries in the first that are real constants, and also the corresponding elements in the second one.

```
sage: from sage.interfaces.tides import subexpressions_list, remove_constants sage:
f(a)=[1+cos(7)*a] sage: l1, l2 = subexpressions_list(f) sage: l1, l2 ([sin(7), cos(7), a*cos(7), a*cos(7)
+ 1], [(‘sin’, 7), (‘cos’, 7), (‘mul’, cos(7), a), (‘add’, 1, a*cos(7))]) sage: remove_constants(l1,l2)
sage: l1, l2 ([a*cos(7), a*cos(7) + 1], [(‘mul’, cos(7), a), (‘add’, 1, a*cos(7))])
```

```
sage.interfaces.tides.remove_repeated(l1, l2)
```

Given two lists, remove the repeated elements in `l1`, and the elements in `l2` that are on the same position. positions.

EXAMPLES:

```
sage: from sage.interfaces.tides import (subexpressions_list, remove_repeated)
sage: f(a)=[1 + a^2, arcsin(a)]
sage: l1, l2 = subexpressions_list(f)
sage: l1, l2
([a^2, a^2 + 1, a^2, -a^2, -a^2 + 1, sqrt(-a^2 + 1), arcsin(a)],
```

(continues on next page)

(continued from previous page)

```

(['mul', a, a),
('add', 1, a^2),
(['mul', a, a),
(['mul', -1, a^2),
('add', 1, -a^2),
('pow', -a^2 + 1, 0.5),
('asin', a)])
sage: remove_repeated(l1, l2)
sage: l1, l2
([a^2, a^2 + 1, -a^2, -a^2 + 1, sqrt(-a^2 + 1), arcsin(a)],
(['mul', a, a),
('add', 1, a^2),
(['mul', -1, a^2),
('add', 1, -a^2),
('pow', -a^2 + 1, 0.5),
('asin', a)])

```

`sage.interfaces.tides.subexpressions_list(f, pars=None)`

Construct the lists with the intermediate steps on the evaluation of the function.

INPUT:

- `f` – a symbolic function of several components.
- `pars` – a list of the parameters that appear in the function this should be the symbolic constants that appear in `f` but are not arguments.

OUTPUT:

- a list of the intermediate subexpressions that appear in the evaluation of `f`.
- a list with the operations used to construct each of the subexpressions. each element of this list is a tuple, formed by a string describing the operation made, and the operands.

For the trigonometric functions, some extra expressions will be added. These extra expressions will be used later to compute their derivatives.

EXAMPLES:

```

sage: from sage.interfaces.tides import subexpressions_list
sage: var('x,y')
(x, y)
sage: f(x,y) = [x^2+y, cos(x)/log(y)]
sage: subexpressions_list(f)
([x^2, x^2 + y, sin(x), cos(x), log(y), cos(x)/log(y)],
(['mul', x, x),
('add', y, x^2),
('sin', x),
('cos', x),
('log', y),
('div', log(y), cos(x))])

```

```

sage: f(a)=[cos(a), arctan(a)]
sage: from sage.interfaces.tides import subexpressions_list
sage: subexpressions_list(f)

```

(continues on next page)

(continued from previous page)

```
([sin(a), cos(a), a^2, a^2 + 1, arctan(a)],
 [('sin', a), ('cos', a), ('mul', a, a), ('add', 1, a^2), ('atan', a)])
```

```
sage: from sage.interfaces.tides import subexpressions_list
sage: var('s,b,r')
(s, b, r)
sage: f(t,x,y,z)= [s*(y-x),x*(r-z)-y,x*y-b*z]
sage: subexpressions_list(f,[s,b,r])
([-y,
 x - y,
 s*(x - y),
 -s*(x - y),
 -z,
 r - z,
 (r - z)*x,
 -y,
 (r - z)*x - y,
 x*y,
 b*z,
 -b*z,
 x*y - b*z],
 [('mul', -1, y),
 ('add', -y, x),
 ('mul', x - y, s),
 ('mul', -1, s*(x - y)),
 ('mul', -1, z),
 ('add', -z, r),
 ('mul', x, r - z),
 ('mul', -1, y),
 ('add', -y, (r - z)*x),
 ('mul', y, x),
 ('mul', z, b),
 ('mul', -1, b*z),
 ('add', -b*z, x*y)])
```

```
sage: var('x, y')
(x, y)
sage: f(x,y)=[exp(x^2+sin(y))]
sage: from sage.interfaces.tides import *
sage: subexpressions_list(f)
([x^2, sin(y), cos(y), x^2 + sin(y), e^(x^2 + sin(y))],
 [('mul', x, x),
 ('sin', y),
 ('cos', y),
 ('add', sin(y), x^2),
 ('exp', x^2 + sin(y))])
```

INTERFACE TO THE SAGE CLEANER

Trivia Note: For the name “sage-cleaner”, think of the “The Cleaner” from Pulp Fiction: <http://www.frankjankowski.de/quiz/illus/keitel.jpg>

```
sage.interfaces.cleaner.start_cleaner()  
    Start sage-cleaner in a new process group.
```


QUITTING INTERFACES

`sage.interfaces.quit.expect_quitall(verbose=False)`

EXAMPLES:

```
sage: sage.interfaces.quit.expect_quitall()
sage: gp.eval('a=10')
'10'
sage: gp('a')
10
sage: sage.interfaces.quit.expect_quitall()
sage: gp('a')
a
sage: sage.interfaces.quit.expect_quitall(verbose=True)
Exiting PARI/GP interpreter with PID ... running .../gp --fast --emacs --quiet --
↳stacksize 100000000
```

`sage.interfaces.quit.invalidate_all()`

Invalidate all of the expect interfaces.

This is used, e.g., by the fork-based `@parallel` decorator.

EXAMPLES:

```
sage: a = maxima(2); b = gp(3)
sage: a, b
(2, 3)
sage: sage.interfaces.quit.invalidate_all()
sage: a
(invalid Maxima object -- The maxima session in which this object was defined is no_
↳longer running.)
sage: b
(invalid PARI/GP interpreter object -- The pari session in which this object was_
↳defined is no longer running.)
```

However the maxima and gp sessions should still work out, though with their state reset:

```
sage: a = maxima(2); b = gp(3) sage: a, b (2, 3)
```

`sage.interfaces.quit.is_running(pid)`

Return True if and only if there is a process with id pid running.

`sage.interfaces.quit.kill_spawned_jobs(verbose=False)`

INPUT:

- `verbose` – bool (default: False); if True, display a message each time a process is sent a kill signal

EXAMPLES:

```
sage: gp.eval('a=10')
'10'
sage: sage.interfaces.quit.kill_spawned_jobs(verbose=False)
sage: sage.interfaces.quit.expect_quitall()
sage: gp.eval('a=10')
'10'
sage: sage.interfaces.quit.kill_spawned_jobs(verbose=True)
Killing spawned job ...
```

After doing the above, we do the following to avoid confusion in other doctests:

```
sage: sage.interfaces.quit.expect_quitall()
```

`sage.interfaces.quit.register_spawned_process(pid, cmd="")`

Write a line to the `spawned_processes` file with the given `pid` and `cmd`.

`sage.interfaces.quit.sage_spawned_process_file()`

EXAMPLES:

```
sage: from sage.interfaces.quit import sage_spawned_process_file
sage: len(sage_spawned_process_file()) > 1
True
```

AN INTERFACE TO READ DATA FILES

`sage.interfaces.read_data.read_data(f, t)`

Read data from file 'f' and class 't' (one element per line), and returns a list of elements.

INPUT:

- 'f' – a file name
- 't' – a class (objects will be coerced to that class)

OUTPUT:

a list of elements of class 't'

EXAMPLES:

```
sage: indata = tmp_filename()
sage: f = open(indata, "w")
sage: _ = f.write("17\n42\n")
sage: f.close()
sage: l = read_data(indata, ZZ); l
[17, 42]
sage: f = open(indata, "w")
sage: _ = f.write("1.234\n5.678\n")
sage: f.close()
sage: l = read_data(indata, RealField(17)); l
[1.234, 5.678]
```


INDICES AND TABLES

- [Index](#)
- [Module Index](#)
- [Search Page](#)

PYTHON MODULE INDEX

i

sage.interfaces.axiom, 15
sage.interfaces.cleaner, 395
sage.interfaces.ecm, 21
sage.interfaces.expect, 9
sage.interfaces.four_ti_2, 27
sage.interfaces.fricas, 33
sage.interfaces.frobby, 45
sage.interfaces.gap, 49
sage.interfaces.gap3, 59
sage.interfaces.gfan, 67
sage.interfaces.giac, 69
sage.interfaces.gnuplot, 79
sage.interfaces.gp, 81
sage.interfaces.interface, 3
sage.interfaces.jmoldata, 91
sage.interfaces.kash, 93
sage.interfaces.kenzo, 103
sage.interfaces.latte, 131
sage.interfaces.lie, 135
sage.interfaces.lisp, 143
sage.interfaces.macaulay2, 147
sage.interfaces.magma, 161
sage.interfaces.magma_free, 181
sage.interfaces.maple, 183
sage.interfaces.mathematica, 191
sage.interfaces.mathics, 201
sage.interfaces.matlab, 213
sage.interfaces.maxima, 219
sage.interfaces.maxima_abstract, 229
sage.interfaces.maxima_lib, 247
sage.interfaces.mupad, 257
sage.interfaces.mwrank, 261
sage.interfaces.octave, 265
sage.interfaces.phc, 271
sage.interfaces.polymake, 279
sage.interfaces.povray, 291
sage.interfaces.psage, 293
sage.interfaces.qepcad, 295
sage.interfaces.qsieve, 321
sage.interfaces.quit, 397
sage.interfaces.r, 325
sage.interfaces.read_data, 399
sage.interfaces.rubik, 339
sage.interfaces.sage0, 343
sage.interfaces.sagespawn, 13
sage.interfaces.scilab, 349
sage.interfaces.singular, 355
sage.interfaces.sympy, 377
sage.interfaces.tachyon, 379
sage.interfaces.tides, 391

Symbols

- `__call__()` (*sage.interfaces.tachyon.TachyonRT method*), 389
 - `_sage_()` (*sage.interfaces.fricas.FriCASElement method*), 39
 - `_sage_()` (*sage.interfaces.macaulay2.Macaulay2Element method*), 153
- ## A
- `A()` (*sage.interfaces.qepcad.qepcad_formula_factory method*), 312
 - `after_print_text()` (*sage.interfaces.macaulay2.Macaulay2Element method*), 155
 - `alexander_dual()` (*sage.interfaces.frobby.Frobby method*), 45
 - `all_but_finitely_many()` (*sage.interfaces.qepcad.qepcad_formula_factory method*), 314
 - `and_()` (*sage.interfaces.qepcad.qepcad_formula_factory method*), 314
 - `answer()` (*sage.interfaces.qepcad.Qepcad method*), 303
 - `application()` (*sage.interfaces.polymake.PolymakeAbstractElementFunction method*), 279
 - `arguments()` (*sage.interfaces.maxima_abstract.MaximaAbstractElementFunction method*), 243
 - `as_type()` (*sage.interfaces.axiom.PanAxiomElement method*), 18
 - `AsciiArtString` (*class in sage.interfaces.interface*), 3
 - `assign_names()` (*sage.interfaces.magma.MagmaElement method*), 174
 - `AssignNames()` (*sage.interfaces.magma.MagmaElement method*), 174
 - `associated_primes()` (*sage.interfaces.frobby.Frobby method*), 46
 - `assume()` (*sage.interfaces.qepcad.Qepcad method*), 303
 - `atomic()` (*sage.interfaces.qepcad.qepcad_formula_factory method*), 314
 - `Attach()` (*sage.interfaces.magma.Magma method*), 165
 - `attach()` (*sage.interfaces.magma.Magma method*), 166
 - `attach_spec()` (*sage.interfaces.magma.Magma method*), 167
 - `AttachSpec()` (*sage.interfaces.magma.Magma method*), 165
 - `attrib()` (*sage.interfaces.singular.SingularElement method*), 368
 - `attribute()` (*sage.interfaces.interface.InterfaceElement method*), 5
 - `available_packages()` (*sage.interfaces.r.R method*), 329
 - `Axiom` (*class in sage.interfaces.axiom*), 17
 - `axiom_console()` (*in module sage.interfaces.axiom*), 19
 - `AxiomElement` (*in module sage.interfaces.axiom*), 17
 - `AxiomExpectFunction` (*in module sage.interfaces.axiom*), 17
 - `AxiomFunctionElement` (*in module sage.interfaces.axiom*), 17
- ## B
- `bar_call()` (*sage.interfaces.magma.Magma method*), 167
 - `basis()` (*sage.interfaces.kenzo.KenzoChainComplex method*), 106
 - `BicomplexSpectralSequence()` (*in module sage.interfaces.kenzo*), 103
 - `blackbox()` (*sage.interfaces.phc.PHC method*), 271
 - `bool()` (*sage.interfaces.fricas.FriCASElement method*), 42
 - `bool()` (*sage.interfaces.interface.InterfaceElement method*), 6
 - `build_morphism()` (*in module sage.interfaces.kenzo*), 128
- ## C
- `C()` (*sage.interfaces.qepcad.qepcad_formula_factory method*), 312
 - `call()` (*sage.interfaces.four_ti_2.FourTi2 method*), 27
 - `call()` (*sage.interfaces.interface.Interface method*), 3
 - `call()` (*sage.interfaces.r.R method*), 330
 - `cartesian_product()` (*sage.interfaces.kenzo.KenzoSimplicialSet method*), 119
 - `cell()` (*sage.interfaces.qepcad.Qepcad method*), 303

- eval() (*sage.interfaces.sage0.Sage method*), 345
 eval() (*sage.interfaces.scilab.Scilab method*), 352
 eval() (*sage.interfaces.singular.Singular method*), 361
 evaluate() (*sage.interfaces.magma.MagmaElement method*), 175
 evaluation() (*sage.interfaces.kenzo.KenzoChainComplexMorphism method*), 113
 exactly_k() (*sage.interfaces.qepcad.qepcad_formula_factory method*), 315
 example() (*sage.interfaces.maxima_abstract.MaximaAbstract method*), 232
 execute() (*sage.interfaces.interface.Interface method*), 4
 exists() (*sage.interfaces.qepcad.qepcad_formula_factory method*), 316
 Expect (*class in sage.interfaces.expect*), 9
 expect() (*sage.interfaces.expect.Expect method*), 10
 expect() (*sage.interfaces.giac.Giac method*), 74
 expect() (*sage.interfaces.maple.Maple method*), 187
 expect() (*sage.interfaces.mupad.Mupad method*), 258
 expect_peek() (*sage.interfaces.sagespawn.SageSpawn method*), 14
 expect_quitall() (*in module sage.interfaces.quit*), 397
 expect_upto() (*sage.interfaces.sagespawn.SageSpawn method*), 14
 ExpectElement (*class in sage.interfaces.expect*), 12
 ExpectFunction (*class in sage.interfaces.expect*), 12
 export_image() (*sage.interfaces.jmoldata.JmolData method*), 91
 extcode_dir() (*in module sage.interfaces.magma*), 179
 external_string() (*sage.interfaces.macaulay2.Macaulay2Element method*), 156
- ## F
- F() (*sage.interfaces.qepcad.qepcad_formula_factory method*), 313
 factor() (*sage.interfaces.ecm.ECM method*), 22
 final_stats() (*sage.interfaces.qepcad.Qepcad method*), 304
 find_factor() (*sage.interfaces.ecm.ECM method*), 23
 flush() (*sage.interfaces.magma.MagmaGBLogPrettyPrinter method*), 179
 flush() (*sage.interfaces.singular.SingularGBLogPrettyPrinter method*), 374
 forall() (*sage.interfaces.qepcad.qepcad_formula_factory method*), 316
 format_cube() (*sage.interfaces.rubik.CubexSolver method*), 339
 format_cube() (*sage.interfaces.rubik.DikSolver method*), 340
 format_cube() (*sage.interfaces.rubik.OptimalSolver method*), 340
 formula() (*sage.interfaces.qepcad.qepcad_formula_factory method*), 316
 FourTi2 (*class in sage.interfaces.four_ti_2*), 27
 FriCAS (*class in sage.interfaces.fricas*), 36
 fricas_console() (*in module sage.interfaces.fricas*), 38
 FriCASElement (*class in sage.interfaces.fricas*), 38
 FriCASExpectFunction (*class in sage.interfaces.fricas*), 43
 FriCASFunctionElement (*class in sage.interfaces.fricas*), 43
 Frobbly (*class in sage.interfaces.frobbly*), 45
 full_typename() (*sage.interfaces.polymake.PolymakeElement method*), 283
 function() (*sage.interfaces.maxima_abstract.MaximaAbstract method*), 233
 function_call() (*sage.interfaces.gap.Gap_generic method*), 54
 function_call() (*sage.interfaces.interface.Interface method*), 4
 function_call() (*sage.interfaces.kash.Kash method*), 100
 function_call() (*sage.interfaces.lie.LiE method*), 140
 function_call() (*sage.interfaces.lisp.Lisp method*), 144
 function_call() (*sage.interfaces.magma.Magma method*), 170
 function_call() (*sage.interfaces.polymake.PolymakeAbstract method*), 281
 function_call() (*sage.interfaces.r.R method*), 331
 FunctionElement (*class in sage.interfaces.expect*), 12
- ## G
- G() (*sage.interfaces.qepcad.qepcad_formula_factory method*), 313
 Gap (*class in sage.interfaces.gap*), 51
 Gap3 (*class in sage.interfaces.gap3*), 64
 gap3_console() (*in module sage.interfaces.gap3*), 65
 gap3_version() (*in module sage.interfaces.gap3*), 66
 GAP3Element (*class in sage.interfaces.gap3*), 62
 GAP3Record (*class in sage.interfaces.gap3*), 63
 gap_command() (*in module sage.interfaces.gap*), 56
 gap_console() (*in module sage.interfaces.gap*), 56
 Gap_generic (*class in sage.interfaces.gap*), 54
 gap_reset_workspace() (*in module sage.interfaces.gap*), 56
 GapElement (*class in sage.interfaces.gap*), 53
 GapElement_generic (*class in sage.interfaces.gap*), 53
 GapFunction (*class in sage.interfaces.gap*), 53
 GapFunctionElement (*class in sage.interfaces.gap*), 54
 gc_disabled (*class in sage.interfaces.expect*), 12
 gen() (*sage.interfaces.fricas.FriCASElement method*), 43

gp_console() (in module *sage.interfaces.gp*), 88
 gp_version() (in module *sage.interfaces.gp*), 88
 GpElement (class in *sage.interfaces.gp*), 87
 graver() (*sage.interfaces.four_ti_2.FourTi2* method), 28
 groebner() (*sage.interfaces.four_ti_2.FourTi2* method), 28
 group() (*sage.interfaces.kenzo.KenzoSpectralSequence* method), 124

H

hasattr() (*sage.interfaces.interface.InterfaceElement* method), 6
 help() (*sage.interfaces.gap.Gap* method), 52
 help() (*sage.interfaces.gap3.Gap3* method), 65
 help() (*sage.interfaces.giac.Giac* method), 75
 help() (*sage.interfaces.gp.Gp* method), 85
 help() (*sage.interfaces.interface.Interface* method), 4
 help() (*sage.interfaces.interface.InterfaceFunctionElement* method), 7
 help() (*sage.interfaces.kash.Kash* method), 100
 help() (*sage.interfaces.lie.LiE* method), 140
 help() (*sage.interfaces.lisp.Lisp* method), 144
 help() (*sage.interfaces.macaulay2.Macaulay2* method), 150
 help() (*sage.interfaces.magma.Magma* method), 171
 help() (*sage.interfaces.maple.Maple* method), 187
 help() (*sage.interfaces.mathematica.Mathematica* method), 197
 help() (*sage.interfaces.mathics.Mathics* method), 208
 help() (*sage.interfaces.maxima_abstract.MaximaAbstract* method), 233
 help() (*sage.interfaces.polymake.PolymakeAbstract* method), 281
 help() (*sage.interfaces.r.R* method), 331
 help() (*sage.interfaces.tachyon.TachyonRT* method), 389
 help_search() (*sage.interfaces.kash.Kash* method), 101
 HelpExpression (class in *sage.interfaces.r*), 329
 hilbert() (*sage.interfaces.four_ti_2.FourTi2* method), 28
 hilbert() (*sage.interfaces.frobby.Frobby* method), 46
 homology() (*sage.interfaces.kenzo.KenzoChainComplex* method), 107
 homotopy_group() (*sage.interfaces.kenzo.KenzoSimplicialSet* method), 120

I

ideal() (*sage.interfaces.macaulay2.Macaulay2* method), 150
 ideal() (*sage.interfaces.magma.Magma* method), 171
 ideal() (*sage.interfaces.magma.MagmaElement* method), 177

ideal() (*sage.interfaces.singular.Singular* method), 363
 identity_morphism() (*sage.interfaces.kenzo.KenzoChainComplex* method), 108
 iff() (*sage.interfaces.qepcad.qepcad_formula_factory* method), 317
 imag() (*sage.interfaces.maxima_abstract.MaximaAbstractElement* method), 239
 implies() (*sage.interfaces.qepcad.qepcad_formula_factory* method), 317
 index() (*sage.interfaces.qepcad.QepcadCell* method), 306
 infinitely_many() (*sage.interfaces.qepcad.qepcad_formula_factory* method), 317
 install_packages() (*sage.interfaces.r.R* method), 332
 integral() (*sage.interfaces.giac.GiacElement* method), 75
 integral() (*sage.interfaces.maxima_abstract.MaximaAbstractElement* method), 240
 integral() (*sage.interfaces.maxima_abstract.MaximaAbstractElementFu* method), 244
 integrate() (in module *sage.interfaces.latte*), 132
 integrate() (*sage.interfaces.giac.GiacElement* method), 76
 integrate() (*sage.interfaces.maxima_abstract.MaximaAbstractElement* method), 240
 integrate() (*sage.interfaces.maxima_abstract.MaximaAbstractElementFu* method), 244
 interact() (*sage.interfaces.ecm.ECM* method), 24
 interact() (*sage.interfaces.gnuplot.Gnuplot* method), 79
 interact() (*sage.interfaces.interface.Interface* method), 4
 Interface (class in *sage.interfaces.interface*), 3
 InterfaceElement (class in *sage.interfaces.interface*), 5
 InterfaceFunction (class in *sage.interfaces.interface*), 7
 InterfaceFunctionElement (class in *sage.interfaces.interface*), 7
 interrupt() (*sage.interfaces.expect.Expect* method), 10
 interrupt() (*sage.interfaces.gap.Gap_generic* method), 55
 intmod_gap_to_sage() (in module *sage.interfaces.gap*), 57
 invalidate_all() (in module *sage.interfaces.quit*), 397
 irreducible_decomposition() (*sage.interfaces.frobby.Frobby* method), 47
 is_AxiomElement() (in module *sage.interfaces.axiom*), 19
 is_ExpectElement() (in module *sage.interfaces.expect*), 12

- load() (*sage.interfaces.maple.Maple method*), 187
load() (*sage.interfaces.singular.Singular method*), 364
load_package() (*sage.interfaces.gap.Gap_generic method*), 55
log() (*sage.interfaces.qsieve.qsieve_nonblock method*), 322
loop_space() (*sage.interfaces.kenzo.KenzoSimplicialSet method*), 121
- ## M
- Macaulay2 (*class in sage.interfaces.macaulay2*), 148
Macaulay2.options() (*in module sage.interfaces.macaulay2*), 150
macaulay2_console() (*in module sage.interfaces.macaulay2*), 158
Macaulay2Element (*class in sage.interfaces.macaulay2*), 152
Macaulay2Function (*class in sage.interfaces.macaulay2*), 158
Macaulay2FunctionElement (*class in sage.interfaces.macaulay2*), 158
Magma (*class in sage.interfaces.magma*), 164
magma_console() (*in module sage.interfaces.magma*), 180
magma_free_eval() (*in module sage.interfaces.magma_free*), 181
magma_gb_standard_options() (*in module sage.interfaces.magma*), 180
MagmaElement (*class in sage.interfaces.magma*), 174
MagmaExpr (*class in sage.interfaces.magma_free*), 181
MagmaFree (*class in sage.interfaces.magma_free*), 181
MagmaFunction (*class in sage.interfaces.magma*), 179
MagmaFunctionElement (*class in sage.interfaces.magma*), 179
MagmaGBDefaultContext (*class in sage.interfaces.magma*), 179
MagmaGBLogPrettyPrinter (*class in sage.interfaces.magma*), 179
make_cells() (*sage.interfaces.qepcad.Qepcad method*), 304
Maple (*class in sage.interfaces.maple*), 186
maple_console() (*in module sage.interfaces.maple*), 189
MapleElement (*class in sage.interfaces.maple*), 189
MapleFunction (*class in sage.interfaces.maple*), 189
MapleFunctionElement (*class in sage.interfaces.maple*), 189
Mathematica (*class in sage.interfaces.mathematica*), 196
mathematica_console() (*in module sage.interfaces.mathematica*), 198
MathematicaElement (*class in sage.interfaces.mathematica*), 197
MathematicaFunction (*class in sage.interfaces.mathematica*), 198
MathematicaFunctionElement (*class in sage.interfaces.mathematica*), 198
Mathics (*class in sage.interfaces.mathics*), 206
mathics_console() (*in module sage.interfaces.mathics*), 210
MathicsElement (*class in sage.interfaces.mathics*), 208
Matlab (*class in sage.interfaces.matlab*), 215
matlab_console() (*in module sage.interfaces.matlab*), 216
matlab_version() (*in module sage.interfaces.matlab*), 217
MatlabElement (*class in sage.interfaces.matlab*), 216
matrix() (*sage.interfaces.kenzo.KenzoSpectralSequence method*), 124
matrix() (*sage.interfaces.singular.Singular method*), 364
max_at_to_sage() (*in module sage.interfaces.maxima_lib*), 252
max_harmonic_to_sage() (*in module sage.interfaces.maxima_lib*), 252
max_pochhammer_to_sage() (*in module sage.interfaces.maxima_lib*), 253
max_to_sr() (*in module sage.interfaces.maxima_lib*), 253
max_to_string() (*in module sage.interfaces.maxima_lib*), 253
Maxima (*class in sage.interfaces.maxima*), 225
maxima_console() (*in module sage.interfaces.maxima_abstract*), 245
maxima_version() (*in module sage.interfaces.maxima_abstract*), 245
MaximaAbstract (*class in sage.interfaces.maxima_abstract*), 229
MaximaAbstractElement (*class in sage.interfaces.maxima_abstract*), 238
MaximaAbstractElementFunction (*class in sage.interfaces.maxima_abstract*), 243
MaximaElement (*class in sage.interfaces.maxima*), 227
MaximaElementFunction (*class in sage.interfaces.maxima*), 227
MaximaLib (*class in sage.interfaces.maxima_lib*), 248
MaximaLibElement (*class in sage.interfaces.maxima_lib*), 250
MaximaLibElementFunction (*class in sage.interfaces.maxima_lib*), 251
mdiff_to_sage() (*in module sage.interfaces.maxima_lib*), 253
methods() (*sage.interfaces.magma.MagmaElement method*), 177
minimize() (*sage.interfaces.four_ti_2.FourTi2 method*), 29
mixed_volume() (*sage.interfaces.phc.PHC method*),

272
 mlist_to_sage() (in *sage.interfaces.maxima_lib*), 254
 module
 sage.interfaces.axiom, 15
 sage.interfaces.cleaner, 395
 sage.interfaces.ecm, 21
 sage.interfaces.expect, 9
 sage.interfaces.four_ti_2, 27
 sage.interfaces.fricas, 33
 sage.interfaces.frobby, 45
 sage.interfaces.gap, 49
 sage.interfaces.gap3, 59
 sage.interfaces.gfan, 67
 sage.interfaces.giac, 69
 sage.interfaces.gnuplot, 79
 sage.interfaces.gp, 81
 sage.interfaces.interface, 3
 sage.interfaces.jmoldata, 91
 sage.interfaces.kash, 93
 sage.interfaces.kenzo, 103
 sage.interfaces.latte, 131
 sage.interfaces.lie, 135
 sage.interfaces.lisp, 143
 sage.interfaces.macaulay2, 147
 sage.interfaces.magma, 161
 sage.interfaces.magma_free, 181
 sage.interfaces.maple, 183
 sage.interfaces.mathematica, 191
 sage.interfaces.mathics, 201
 sage.interfaces.matlab, 213
 sage.interfaces.maxima, 219
 sage.interfaces.maxima_abstract, 229
 sage.interfaces.maxima_lib, 247
 sage.interfaces.mupad, 257
 sage.interfaces.mwrank, 261
 sage.interfaces.octave, 265
 sage.interfaces.phc, 271
 sage.interfaces.polymake, 279
 sage.interfaces.povray, 291
 sage.interfaces.psage, 293
 sage.interfaces.qepcad, 295
 sage.interfaces.qsieve, 321
 sage.interfaces.quit, 397
 sage.interfaces.r, 325
 sage.interfaces.read_data, 399
 sage.interfaces.rubik, 339
 sage.interfaces.sage0, 343
 sage.interfaces.sagespawn, 13
 sage.interfaces.scilab, 349
 sage.interfaces.singular, 355
 sage.interfaces.sympy, 377
 sage.interfaces.tachyon, 379
 sage.interfaces.tides, 391

MooreSpace() (in module *sage.interfaces.kenzo*), 125
 morphism_dictmat() (in module *sage.interfaces.kenzo*), 129
 mqapply_to_sage() (in module *sage.interfaces.maxima_lib*), 254
 mrat_to_sage() (in module *sage.interfaces.maxima_lib*), 254
 Mupad (class in *sage.interfaces.mupad*), 257
 mupad_console() (in module *sage.interfaces.mupad*), 258
 MupadElement (class in *sage.interfaces.mupad*), 258
 MupadFunction (class in *sage.interfaces.mupad*), 258
 MupadFunctionElement (class in *sage.interfaces.mupad*), 258
 Mwrnk() (in module *sage.interfaces.mwrank*), 261
 Mwrnk_class (class in *sage.interfaces.mwrank*), 261
 mwrank_console() (in module *sage.interfaces.mwrank*), 262

N

n() (*sage.interfaces.mathematica.MathematicaElement* method), 197
 n() (*sage.interfaces.mathics.MathicsElement* method), 209
 n() (*sage.interfaces.qsieve.qsieve_nonblock* method), 322
 na() (*sage.interfaces.r.R* method), 332
 name() (*sage.interfaces.interface.Interface* method), 4
 name() (*sage.interfaces.interface.InterfaceElement* method), 6
 name() (*sage.interfaces.macaulay2.Macaulay2Element* method), 156
 new() (*sage.interfaces.interface.Interface* method), 4
 new() (*sage.interfaces.sage0.Sage* method), 345
 new_from() (*sage.interfaces.macaulay2.Macaulay2* method), 150
 new_object() (*sage.interfaces.polymake.PolymakeAbstract* method), 282
 new_with_bits_prec() (*sage.interfaces.gp.Gp* method), 85
 nintegral() (*sage.interfaces.maxima_abstract.MaximaAbstractElement* method), 241
 not_() (*sage.interfaces.qepcad.qepcad_formula_factory* method), 318
 null_morphism() (*sage.interfaces.kenzo.KenzoChainComplex* method), 108
 number_of_children() (*sage.interfaces.qepcad.QepcadCell* method), 307
 numer() (*sage.interfaces.maxima_abstract.MaximaAbstractElement* method), 242

O

objgens() (*sage.interfaces.magma.Magma* method),

- 172
- Octave (*class in sage.interfaces.octave*), 267
- octave_console() (*in module sage.interfaces.octave*), 270
- OctaveElement (*class in sage.interfaces.octave*), 270
- one_curve() (*sage.interfaces.ecm.ECM method*), 24
- op() (*sage.interfaces.maple.MapleElement method*), 189
- operations() (*sage.interfaces.gap3.GAP3Record method*), 63
- opposite() (*sage.interfaces.kenzo.KenzoChainComplexMorphism method*), 114
- OptimalSolver (*class in sage.interfaces.rubik*), 340
- option() (*sage.interfaces.singular.Singular method*), 365
- or_() (*sage.interfaces.qepcad.qepcad_formula_factory method*), 318
- orgn() (*sage.interfaces.kenzo.KenzoChainComplex method*), 109
- ## P
- pairing() (*in module sage.interfaces.kenzo*), 129
- PanAxiom (*class in sage.interfaces.axiom*), 17
- PanAxiomElement (*class in sage.interfaces.axiom*), 18
- PanAxiomExpectFunction (*class in sage.interfaces.axiom*), 19
- PanAxiomFunctionElement (*class in sage.interfaces.axiom*), 19
- parse_max_string() (*in module sage.interfaces.maxima_lib*), 255
- parse_moutput_from_json() (*in module sage.interfaces.mathematica*), 198
- partial_fraction_decomposition() (*sage.interfaces.maxima_abstract.MaximaAbstract method*), 242
- path() (*sage.interfaces.expect.Expect method*), 10
- path_track() (*sage.interfaces.phc.PHC method*), 272
- phase() (*sage.interfaces.qepcad.Qepcad method*), 304
- PHC (*class in sage.interfaces.phc*), 271
- PHC_Object (*class in sage.interfaces.phc*), 274
- pid() (*sage.interfaces.expect.Expect method*), 10
- pid() (*sage.interfaces.qsieve.qsieve_nonblock method*), 322
- plot() (*sage.interfaces.gnuplot.Gnuplot method*), 79
- plot() (*sage.interfaces.r.R method*), 332
- plot2d() (*sage.interfaces.maxima_abstract.MaximaAbstract method*), 234
- plot2d_parametric() (*sage.interfaces.maxima_abstract.MaximaAbstract method*), 234
- plot3d() (*sage.interfaces.gnuplot.Gnuplot method*), 79
- plot3d() (*sage.interfaces.maxima_abstract.MaximaAbstract method*), 235
- plot3d_parametric() (*sage.interfaces.gnuplot.Gnuplot method*), 79
- plot3d_parametric() (*sage.interfaces.maxima_abstract.MaximaAbstract method*), 235
- plot_list() (*sage.interfaces.maxima_abstract.MaximaAbstract method*), 236
- plot_multilist() (*sage.interfaces.maxima_abstract.MaximaAbstract method*), 236
- plot_paths_2d() (*sage.interfaces.phc.PHC method*), 273
- png() (*sage.interfaces.r.R method*), 334
- Polymake (*in module sage.interfaces.polymake*), 279
- polymake_console() (*in module sage.interfaces.polymake*), 290
- PolymakeAbstract (*class in sage.interfaces.polymake*), 279
- PolymakeElement (*class in sage.interfaces.polymake*), 283
- PolymakeError, 286
- PolymakeFunctionElement (*class in sage.interfaces.polymake*), 287
- PolymakeJuPyMake (*class in sage.interfaces.polymake*), 287
- POVRay (*class in sage.interfaces.povray*), 291
- ppi() (*sage.interfaces.four_ti_2.FourTi2 method*), 29
- preparse() (*sage.interfaces.sage0.Sage method*), 345
- PSage (*class in sage.interfaces.psage*), 293
- PSageElement (*class in sage.interfaces.psage*), 294
- pyobject_to_max() (*in module sage.interfaces.maxima_lib*), 255
- ## Q
- Qepcad (*class in sage.interfaces.qepcad*), 303
- qepcad() (*in module sage.interfaces.qepcad*), 309
- qepcad_banner() (*in module sage.interfaces.qepcad*), 311
- qepcad_console() (*in module sage.interfaces.qepcad*), 311
- Qepcad_expect (*class in sage.interfaces.qepcad*), 308
- qepcad_formula_factory (*class in sage.interfaces.qepcad*), 312
- qepcad_version() (*in module sage.interfaces.qepcad*), 319
- QepcadCell (*class in sage.interfaces.qepcad*), 306
- QepcadFunction (*class in sage.interfaces.qepcad*), 308
- qformula (*class in sage.interfaces.qepcad*), 319
- qsieve() (*in module sage.interfaces.qsieve*), 321
- qsieve_block() (*in module sage.interfaces.qsieve*), 322
- qsieve_nonblock (*class in sage.interfaces.qsieve*), 322
- qsolve() (*sage.interfaces.four_ti_2.FourTi2 method*), 29
- qualified_typename() (*sage.interfaces.polymake.PolymakeElement method*), 286

module, 15
 sage.interfaces.cleaner
 module, 395
 sage.interfaces.ecm
 module, 21
 sage.interfaces.expect
 module, 9
 sage.interfaces.four_ti_2
 module, 27
 sage.interfaces.fricas
 module, 33
 sage.interfaces.frobby
 module, 45
 sage.interfaces.gap
 module, 49
 sage.interfaces.gap3
 module, 59
 sage.interfaces.gfan
 module, 67
 sage.interfaces.giac
 module, 69
 sage.interfaces.gnuplot
 module, 79
 sage.interfaces.gp
 module, 81
 sage.interfaces.interface
 module, 3
 sage.interfaces.jmoldata
 module, 91
 sage.interfaces.kash
 module, 93
 sage.interfaces.kenzo
 module, 103
 sage.interfaces.latte
 module, 131
 sage.interfaces.lie
 module, 135
 sage.interfaces.lisp
 module, 143
 sage.interfaces.macaulay2
 module, 147
 sage.interfaces.magma
 module, 161
 sage.interfaces.magma_free
 module, 181
 sage.interfaces.maple
 module, 183
 sage.interfaces.mathematica
 module, 191
 sage.interfaces.mathics
 module, 201
 sage.interfaces.matlab
 module, 213
 sage.interfaces.maxima
 module, 219
 sage.interfaces.maxima_abstract
 module, 229
 sage.interfaces.maxima_lib
 module, 247
 sage.interfaces.mupad
 module, 257
 sage.interfaces.mwrank
 module, 261
 sage.interfaces.octave
 module, 265
 sage.interfaces.phc
 module, 271
 sage.interfaces.polymake
 module, 279
 sage.interfaces.povray
 module, 291
 sage.interfaces.psage
 module, 293
 sage.interfaces.qepcad
 module, 295
 sage.interfaces.qsieve
 module, 321
 sage.interfaces.quit
 module, 397
 sage.interfaces.r
 module, 325
 sage.interfaces.read_data
 module, 399
 sage.interfaces.rubik
 module, 339
 sage.interfaces.sage0
 module, 343
 sage.interfaces.sagespawn
 module, 13
 sage.interfaces.scilab
 module, 349
 sage.interfaces.singular
 module, 355
 sage.interfaces.sympy
 module, 377
 sage.interfaces.tachyon
 module, 379
 sage.interfaces.tides
 module, 391
 sage0_console() (*in module sage.interfaces.sage0*),
 346
 sage0_version() (*in module sage.interfaces.sage0*),
 346
 sage2matlab_matrix_string()
 (*sage.interfaces.matlab.Matlab* *method*),
 216
 sage2octave_matrix_string()
 (*sage.interfaces.octave.Octave* *method*),

- 268
- sage2scilab_matrix_string()
 - (*sage.interfaces.scilab.Scilab* method), 352
- sage_flattened_str_list()
 - (*sage.interfaces.singular.SingularElement* method), 369
- SAGE_GAP_COMMAND, 51
- SAGE_GAP_MEMORY, 51
- sage_global_ring() (*sage.interfaces.singular.SingularElement* method), 369
- sage_matrix() (*sage.interfaces.singular.SingularElement* method), 371
- sage_poly() (*sage.interfaces.singular.SingularElement* method), 371
- sage_polystring() (*sage.interfaces.macaulay2.Macaulay2Element* method), 157
- sage_polystring() (*sage.interfaces.singular.SingularElement* method), 372
- sage_rat() (in module *sage.interfaces.maxima_lib*), 256
- sage_spawned_process_file() (in module *sage.interfaces.quit*), 398
- sage_structured_str_list()
 - (*sage.interfaces.singular.SingularElement* method), 372
- SageElement (class in *sage.interfaces.sage0*), 346
- SageFunction (class in *sage.interfaces.sage0*), 346
- SagePtyProcess (class in *sage.interfaces.sagespawn*), 13
- SageSpawn (class in *sage.interfaces.sagespawn*), 14
- sample_point() (*sage.interfaces.qepcad.QepcadCell* method), 307
- sample_point_dict()
 - (*sage.interfaces.qepcad.QepcadCell* method), 307
- save_as_start() (*sage.interfaces.phc.PHC_Object* method), 275
- save_image() (*sage.interfaces.mathematica.MathematicaElement* method), 197
- save_image() (*sage.interfaces.mathics.MathicsElement* method), 209
- save_workspace() (*sage.interfaces.gap.Gap* method), 53
- SChainComplex() (in module *sage.interfaces.kenzo*), 126
- Scilab (class in *sage.interfaces.scilab*), 351
- scilab_console() (in module *sage.interfaces.scilab*), 353
- scilab_version() (in module *sage.interfaces.scilab*), 354
- ScilabElement (class in *sage.interfaces.scilab*), 353
- serre_spectral_sequence()
 - (*sage.interfaces.kenzo.KenzoSimplicialSet* method), 121
- server() (*sage.interfaces.expect.Expect* method), 11
- set() (*sage.interfaces.axiom.PanAxiom* method), 18
- set() (*sage.interfaces.fricas.FriCAS* method), 38
- set() (*sage.interfaces.gap.Gap* method), 53
- set() (*sage.interfaces.giac.Giac* method), 75
- set() (*sage.interfaces.gp.Gp* method), 85
- set() (*sage.interfaces.interface.Interface* method), 5
- set() (*sage.interfaces.kash.Kash* method), 101
- set() (*sage.interfaces.lie.LiE* method), 141
- set() (*sage.interfaces.lisp.Lisp* method), 145
- set() (*sage.interfaces.macaulay2.Macaulay2* method), 152
- set() (*sage.interfaces.magma.Magma* method), 173
- set() (*sage.interfaces.maple.Maple* method), 188
- set() (*sage.interfaces.mathematica.MathematicaElement* method), 197
- set() (*sage.interfaces.mathics.Mathics* method), 208
- set() (*sage.interfaces.matlab.Matlab* method), 216
- set() (*sage.interfaces.matlab.MatlabElement* method), 216
- set() (*sage.interfaces.maxima.Maxima* method), 226
- set() (*sage.interfaces.maxima_lib.MaximaLib* method), 250
- set() (*sage.interfaces.mupad.Mupad* method), 258
- set() (*sage.interfaces.octave.Octave* method), 269
- set() (*sage.interfaces.polymake.PolymakeAbstract* method), 282
- set() (*sage.interfaces.p sage.PSage* method), 294
- set() (*sage.interfaces.r.R* method), 335
- set() (*sage.interfaces.sage0.Sage* method), 346
- set() (*sage.interfaces.scilab.Scilab* method), 352
- set() (*sage.interfaces.scilab.ScilabElement* method), 353
- set() (*sage.interfaces.singular.Singular* method), 366
- set_default() (*sage.interfaces.gp.Gp* method), 86
- set_magma_attribute()
 - (*sage.interfaces.magma.MagmaElement* method), 178
- set_nthreads() (*sage.interfaces.magma.Magma* method), 173
- set_precision() (*sage.interfaces.gp.Gp* method), 86
- set_real_precision() (*sage.interfaces.gp.Gp* method), 86
- set_ring() (*sage.interfaces.singular.Singular* method), 367
- set_ring() (*sage.interfaces.singular.SingularElement* method), 373
- set_seed() (*sage.interfaces.gap.Gap* method), 53
- set_seed() (*sage.interfaces.gp.Gp* method), 87
- set_seed() (*sage.interfaces.interface.Interface* method), 5
- set_seed() (*sage.interfaces.macaulay2.Macaulay2* method), 152

- set_seed() (*sage.interfaces.magma.Magma* method), 173
- set_seed() (*sage.interfaces.maxima.Maxima* method), 227
- set_seed() (*sage.interfaces.octave.Octave* method), 269
- set_seed() (*sage.interfaces.r.R* method), 335
- set_seed() (*sage.interfaces.scilab.Scilab* method), 352
- set_seed() (*sage.interfaces.singular.Singular* method), 367
- set_series_precision() (*sage.interfaces.gp.Gp* method), 87
- set_server_and_command() (*sage.interfaces.expect.Expect* method), 11
- set_truth() (*sage.interfaces.qepcad.QepcadCell* method), 308
- set_truth_value() (*sage.interfaces.qepcad.Qepcad* method), 305
- set_verbose() (*sage.interfaces.magma.Magma* method), 173
- SetNthreads() (*sage.interfaces.magma.Magma* method), 166
- setring() (*sage.interfaces.singular.Singular* method), 367
- SetVerbose() (*sage.interfaces.magma.Magma* method), 166
- SFiniteSimplicialSet() (in module *sage.interfaces.kenzo*), 127
- sharp() (*sage.interfaces.macaulay2.Macaulay2Element* method), 157
- show() (*sage.interfaces.mathematica.MathematicaElement* method), 197
- show() (*sage.interfaces.mathics.MathicsElement* method), 210
- signs() (*sage.interfaces.qepcad.QepcadCell* method), 308
- SingNot (class in *sage.interfaces.rubik*), 340
- Singular (class in *sage.interfaces.singular*), 360
- singular_console() (in module *sage.interfaces.singular*), 375
- singular_gb_standard_options() (in module *sage.interfaces.singular*), 375
- singular_version() (in module *sage.interfaces.singular*), 375
- SingularElement (class in *sage.interfaces.singular*), 368
- SingularError, 373
- SingularFunction (class in *sage.interfaces.singular*), 373
- SingularFunctionElement (class in *sage.interfaces.singular*), 373
- SingularGBDefaultContext (class in *sage.interfaces.singular*), 373
- SingularGBLogPrettyPrinter (class in *sage.interfaces.singular*), 374
- smash_product() (*sage.interfaces.kenzo.KenzoSimplicialSet* method), 122
- solution_dicts() (*sage.interfaces.phc.PHC_Object* method), 275
- solution_extension() (*sage.interfaces.qepcad.Qepcad* method), 305
- solutions() (*sage.interfaces.phc.PHC_Object* method), 275
- solve() (*sage.interfaces.rubik.CubexSolver* method), 339
- solve() (*sage.interfaces.rubik.DikSolver* method), 340
- solve() (*sage.interfaces.rubik.OptimalSolver* method), 340
- solve_linear() (*sage.interfaces.maxima_abstract.MaximaAbstract* method), 237
- solve_linear_system() (*sage.interfaces.octave.Octave* method), 269
- source() (*sage.interfaces.maple.Maple* method), 188
- source() (*sage.interfaces.r.R* method), 335
- source_complex() (*sage.interfaces.kenzo.KenzoChainComplexMorphism* method), 115
- Sphere() (in module *sage.interfaces.kenzo*), 127
- sr_integral() (*sage.interfaces.maxima_lib.MaximaLib* method), 250
- sr_limit() (*sage.interfaces.maxima_lib.MaximaLib* method), 250
- sr_prod() (*sage.interfaces.maxima_lib.MaximaLib* method), 250
- sr_sum() (*sage.interfaces.maxima_lib.MaximaLib* method), 250
- sr_tlimit() (*sage.interfaces.maxima_lib.MaximaLib* method), 250
- sr_to_max() (in module *sage.interfaces.maxima_lib*), 256
- starstar() (*sage.interfaces.macaulay2.Macaulay2Element* method), 157
- start() (*sage.interfaces.rubik.OptimalSolver* method), 340
- start_cleaner() (in module *sage.interfaces.cleaner*), 395
- start_from() (*sage.interfaces.phc.PHC* method), 273
- stat_model() (*sage.interfaces.r.RElement* method), 336
- stdout_to_string() (in module *sage.interfaces.maxima_lib*), 256
- StdOutContext (class in *sage.interfaces.expect*), 12
- stop() (*sage.interfaces.rubik.OptimalSolver* method), 340
- str() (*sage.interfaces.gap.GapElement* method), 53
- str() (*sage.interfaces.mathematica.MathematicaElement* method), 198
- str() (*sage.interfaces.maxima_abstract.MaximaAbstractElement* method), 198

- method), 242
 string() (*sage.interfaces.singular.Singular* method), 368
 strip_answer() (*sage.interfaces.matlab.Matlab* method), 216
 structure_sheaf() (*sage.interfaces.macaulay2.Macaulay2Element* method), 157
 sub() (*sage.interfaces.magma.MagmaElement* method), 178
 subexpressions_list() (in module *sage.interfaces.tides*), 393
 subs() (*sage.interfaces.macaulay2.Macaulay2Element* method), 157
 subst() (*sage.interfaces.maxima_abstract.MaximaAbstractElement* method), 243
 substitute() (*sage.interfaces.macaulay2.Macaulay2Element* method), 158
 substract() (*sage.interfaces.kenzo.KenzoChainComplexMorphism* method), 116
 sum() (*sage.interfaces.giac.GiacElement* method), 76
 sum() (*sage.interfaces.kenzo.KenzoChainComplexMorphism* method), 117
 suspension() (*sage.interfaces.kenzo.KenzoSimplicialSet* method), 122
 sw_spectral_sequence() (*sage.interfaces.kenzo.KenzoSimplicialSet* method), 123
 symbolic_expression_from_mathematica_string() (in module *sage.interfaces.mathematica*), 199
 sympy_init() (in module *sage.interfaces.sympy*), 378
 sympy_set_to_list() (in module *sage.interfaces.sympy*), 378
- T**
- table() (*sage.interfaces.kenzo.KenzoSpectralSequence* method), 125
 TachyonRT (class in *sage.interfaces.tachyon*), 388
 target_complex() (*sage.interfaces.kenzo.KenzoChainComplexMorphism* method), 118
 temp_project() (*sage.interfaces.four_ti_2.FourTi2* method), 30
 tensor_product() (*sage.interfaces.kenzo.KenzoChainComplexMorphism* method), 109
 terminate_async() (*sage.interfaces.sagespwn.SagePtyProcess* method), 13
 test_all() (in module *sage.interfaces.sympy*), 378
 tilde() (*sage.interfaces.r.RElement* method), 337
 time() (*sage.interfaces.ecm.ECM* method), 25
 time() (*sage.interfaces.qsieve.qsieve_nonblock* method), 323
 to_complex() (in module *sage.interfaces.octave*), 270
 to_latte_polynomial() (in module *sage.interfaces.latte*), 133
 to_poly_solve() (*sage.interfaces.maxima_lib.MaximaLibElement* method), 251
 to_sage() (*sage.interfaces.macaulay2.Macaulay2Element* method), 158
 type() (*sage.interfaces.axiom.PanAxiomElement* method), 18
 type() (*sage.interfaces.lie.LiEElement* method), 141
 type() (*sage.interfaces.singular.SingularElement* method), 373
 typename() (*sage.interfaces.polymake.PolymakeElement* method), 286
 typeof() (*sage.interfaces.polymake.PolymakeElement* method), 286
- U**
- unapply() (*sage.interfaces.giac.GiacElement* method), 76
 unbind() (*sage.interfaces.gap.Gap_generic* method), 55
 UndefinedSageHelper (class in *sage.interfaces.sympy*), 377
 underscore() (*sage.interfaces.macaulay2.Macaulay2Element* method), 158
 unit_quadratic_integer() (*sage.interfaces.maxima_abstract.MaximaAbstract* method), 237
 unparsed_input_form() (*sage.interfaces.axiom.PanAxiomElement* method), 18
 usage() (*sage.interfaces.povray.POVRay* method), 291
 usage() (*sage.interfaces.tachyon.TachyonRT* method), 389
 use() (*sage.interfaces.macaulay2.Macaulay2* method), 152
 user_dir() (*sage.interfaces.expect.Expect* method), 11
- V**
- validate_mwrank_input() (in module *sage.interfaces.mwrank*), 262
 variable_list() (*sage.interfaces.phc.PHC_Object* method), 276
 version() (*sage.interfaces.gap.Gap_generic* method), 56
 version() (*sage.interfaces.giac.Giac* method), 75
 version() (*sage.interfaces.gp.Gp* method), 87
 version() (*sage.interfaces.kash.Kash* method), 101
 version() (*sage.interfaces.lie.LiE* method), 141
 version() (*sage.interfaces.lisp.Lisp* method), 145
 version() (*sage.interfaces.macaulay2.Macaulay2* method), 152
 version() (*sage.interfaces.magma.Magma* method), 174
 version() (*sage.interfaces.matlab.Matlab* method), 216
 version() (*sage.interfaces.maxima_abstract.MaximaAbstract* method), 237
 version() (*sage.interfaces.octave.Octave* method), 270

`version()` (*sage.interfaces.polymake.PolymakeAbstract*
method), 283

`version()` (*sage.interfaces.r.R method*), 336

`version()` (*sage.interfaces.sage0.Sage method*), 346

`version()` (*sage.interfaces.scilab.Scilab method*), 353

`version()` (*sage.interfaces.singular.Singular method*),
368

W

`wedge()` (*sage.interfaces.kenzo.KenzoSimplicialSet*
method), 123

`whos()` (*sage.interfaces.matlab.Matlab method*), 216

`whos()` (*sage.interfaces.scilab.Scilab method*), 353

`with_package()` (*sage.interfaces.maple.Maple*
method), 188

`write()` (*sage.interfaces.magma.MagmaGBLogPrettyPrinter*
method), 179

`write()` (*sage.interfaces.singular.SingularGBLogPrettyPrinter*
method), 374

`write_array()` (*sage.interfaces.four_ti_2.FourTi2*
method), 30

`write_matrix()` (*sage.interfaces.four_ti_2.FourTi2*
method), 30

`write_single_row()` (*sage.interfaces.four_ti_2.FourTi2*
method), 31

X

`X()` (*sage.interfaces.qepcad.qepcad_formula_factory*
method), 313

Z

`zsolve()` (*sage.interfaces.four_ti_2.FourTi2 method*),
31