Interpreter Interfaces

Release 10.6

The Sage Development Team

Apr 01, 2025

CONTENTS

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).

CHAPTER

ONE

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: `WithEqualityById`, `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.

Likewise, the method `_eval_line()` for evaluation of a single line, often makes sense to be overridden.

execute(**args*, ***kwds*)

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

EXAMPLES:

```
sage: maxima.quad_qags(x, x, 0, 1, epsrel=1e-4)
[0.5,5.5511151231257...e-15,21,0]
sage: maxima.function_call('quad_qags', [x, x, 0, 1], {'epsrel':'1e-4'})
[0.5,5.5511151231257...e-15,21,0]
```

```
>>> from sage.all import *
>>> maxima.quad_qags(x, x, Integer(0), Integer(1), epsrel=RealNumber('1e-4'))
[0.5,5.5511151231257...e-15,21,0]
>>> maxima.function_call('quad_qags', [x, x, Integer(0), Integer(1)], {'epsrel
→':'1e-4'})
[0.5,5.5511151231257...e-15,21,0]
```

get(*var*)

Get the value of the variable var.

Note that this needs to be overridden in some interfaces, namely when getting the string representation of an object requires an explicit print command.

get_seed()

Return the seed used to set the random number generator in this interface.

The seed is initialized as `None` but should be set when the interface starts.

EXAMPLES:

```
sage: s = Singular()
sage: s.set_seed(107)
107
sage: s.get_seed()
107
```

```
>>> from sage.all import *
>>> s = Singular()
>>> s.set_seed(Integer(107))
107
>>> s.get_seed()
107
```

get_using_file(*var*)

Return the string representation of the variable var in self, possibly using a file. Use this if var has a huge string representation, since it may be way faster.

Warning

In fact unless a special derived class implements this, it will *not* be any faster. This is the case for this class if you're reading it through introspection and seeing this.

help(*s*)

interact()

This allows you to interactively interact with the child interpreter.

Press `Ctrl` + `D` or type 'quit' or 'exit' to exit and return to Sage.

Note

This is completely different than the console() member function. The console function opens a new copy of the child interpreter, whereas the interact function gives you interactive access to the interpreter that is being used by Sage. Use sage(xxx) or interpretername(xxx) to pull objects in from sage to the interpreter.

name(*new_name=None*)

new(*code*)

rand_seed()

Return a random seed that can be put into `set_seed` function for any interpreter.

This should be overridden if the particular interface needs something other than a small positive integer.

EXAMPLES:

```
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
```

```
>>> from sage.all import *
>>> from sage.interfaces.interface import Interface
>>> i = Interface("")
>>> i.rand_seed() # random
318491487

>>> s = Singular()
>>> 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()
```

(continues on next page)

(continued from previous page)

```
sage: octave.read(filename)  # optional - octave
sage: octave.get('x')        # optional - octave
' 2'
sage: import os
sage: os.unlink(filename)
```

```
>>> from sage.all import *
>>> filename = tmp_filename()
>>> f = open(filename, 'w')
>>> _ = f.write('x = 2\n')
>>> f.close()
>>> octave.read(filename)  # optional - octave
>>> octave.get('x')        # optional - octave
' 2'
>>> import os
>>> 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
```

```
>>> from sage.all import *
>>> s = Singular()
>>> s.set_seed(Integer(1))
1
>>> [s.random(Integer(1),Integer(10)) for i in range(Integer(5))]
[8, 10, 4, 9, 1]

>>> from sage.interfaces.interface import Interface
>>> i = Interface("")
>>> i.set_seed()
```

(continues on next page)

(continued from previous page)

```
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: 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: g = gap('SO(1,4,7)')
sage: k = g.InvariantQuadraticForm()
sage: k.attribute('matrix')
[ [ 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7) ], [ 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), Z(7), 0*Z(7) ], [ 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^0 ] ]
```

```
>>> from sage.all import *
>>> g = gap('SO(1,4,7)')
>>> k = g.InvariantQuadraticForm()
>>> k.attribute('matrix')
[ [ 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7) ], [ 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), Z(7), 0*Z(7) ], [ 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^0 ] ]
```

```
sage: e = gp('ellinit([0,-1,1,-10,-20])')
sage: e.attribute('j')
-122023936/161051
```

```
>>> from sage.all import *
>>> e = gp('ellinit([0,-1,1,-10,-20])')
>>> e.attribute('j')
-122023936/161051
```

bool()

Convert this element to a boolean.

EXAMPLES:

```
sage: singular(0).bool()
False
sage: singular(1).bool()
True
```

```
>>> from sage.all import *
>>> singular(Integer(0)).bool()
False
>>> singular(Integer(1)).bool()
True
```

gen(*n*)

get_using_file()

Return this element's string representation using a file. Use this if `self` has a huge string representation. It'll be way faster.

EXAMPLES:

```
sage: a = maxima(str(2^1000))
sage: a.get_using_file()
'10715086071862673209484250490600018105614048117055336074437503883703510511249
↪36122493198378815695858127594672917553146825187145285692314043598457757469857
↪48039345677748242309854210746050623711418779541821530464749835819412673987675
↪5591655439460770629145711964776865421676604298316526243868372056680693 76'
```

```
>>> from sage.all import *
>>> a = maxima(str(Integer(2)**Integer(1000)))
>>> a.get_using_file()
'10715086071862673209484250490600018105614048117055336074437503883703510511249
↪36122493198378815695858127594672917553146825187145285692314043598457757469857
↪48039345677748242309854210746050623711418779541821530464749835819412673987675
↪5591655439460770629145711964776865421676604298316526243868372056680693 76'
```

hasattr(*attrname*)

Return whether the given attribute is already defined by this object, and in particular is not dynamically generated.

EXAMPLES:

```
sage: m = maxima('2')
sage: m.hasattr('integral')
True
sage: m.hasattr('gcd')
False
```

```
>>> from sage.all import *
>>> m = maxima('2')
>>> m.hasattr('integral')
True
>>> m.hasattr('gcd')
False
```

is_string()

Tell whether this element is a string.

By default, the answer is negative.

name(*new_name=None*)

Return the name of `self`. If `new_name` is passed in, then this function returns a new object identical to `self` whose name is `new_name`.

Note that this can overwrite existing variables in the system.

EXAMPLES:

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

```
>>> from sage.all import *
>>> # optional - rpy2
>>> x = r([Integer(1),Integer(2),Integer(3)]); x
[1] 1 2 3
>>> x.name()
'sage...'
>>> x = r([Integer(1),Integer(2),Integer(3)]).name('x'); x
[1] 1 2 3
>>> x.name()
'x'
```

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

```
>>> from sage.all import *
>>> s5 = gap.SymmetricGroup(Integer(5)).name('s5')
>>> s5
SymmetricGroup( [ 1 .. 5 ] )
>>> s5.name()
's5'
```

sage(**args*, ***kwds*)

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]
```

```
>>> from sage.all import *
>>> gp(Integer(1)/Integer(2)).sage()
```

(continues on next page)

(continued from previous page)

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

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

Bases: SageObject

Interface function.

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

Bases: SageObject

Interface function element.

help()

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

Return True if x is of type *InterfaceElement*.

EXAMPLES:

```
sage: from sage.interfaces.interface import is_InterfaceElement
sage: is_InterfaceElement(2)
doctest:...: DeprecationWarning: the function is_InterfaceElement is deprecated;␣
↪use isinstance(x, sage.interfaces.abc.InterfaceElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
```

```
>>> from sage.all import *
>>> from sage.interfaces.interface import is_InterfaceElement
>>> is_InterfaceElement(Integer(2))
doctest:...: DeprecationWarning: the function is_InterfaceElement is deprecated;␣
↪use isinstance(x, sage.interfaces.abc.InterfaceElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
```

CHAPTER

TWO

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 Issue #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: *Interface*

Expect interface object.

clear_prompts()

command()

Return the command used in this interface as a string.

EXAMPLES:

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

```
>>> from sage.all import *
>>> magma.set_server_and_command(command='magma-2.19')
>>> 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
```

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

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

```
sage: maxima.detach()
```

```
>>> from sage.all import *
>>> maxima.detach()
```

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

INPUT:

- `code` – text to evaluate
- `strip` – boolean; 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` – boolean (default: `True`); if `True` and `code` exceeds an interface-specific threshold then `code` will be communicated via a temporary file rather that 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
```

```
>>> from sage.all import *
>>> pid = gap.pid()
>>> gap.eval('quit;')
''
>>> pid == gap.pid()
True
>>> gap.quit()
>>> pid == gap.pid()
False
```

quit(*verbose=False*)

Quit the running subprocess.

INPUT:

- `verbose` – boolean (default: `False`); whether to print a message when quitting the process

EXAMPLES:

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

```
>>> from sage.all import *
>>> a = maxima('y')
>>> maxima.quit(verbose=True)
Exiting Maxima with PID ... running ...maxima...
>>> 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)
```

```
>>> from sage.all import *
>>> maxima.quit(verbose=True)
```

server()

Return 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'
```

```
>>> from sage.all import *
>>> magma.set_server_and_command(server='remote')
No remote temporary directory (option server_tmpdir) specified, using /tmp/␣
↪on remote
>>> 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 `RuntimeError` if the interface is already started.

INPUT:

- `server` – string or `None` (default); name of a remote host to connect to using `ssh`
- `command` – one of:
  - a string; command line passed to the shell
  - a sequence of an `Executable` and strings, arguments to pass to the executable.

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
```

(continues on next page)

(continued from previous page)

```
sage: magma.server()
'remote'
sage: magma.command()
'ssh -t remote mymagma'
```

```
>>> from sage.all import *
>>> magma.set_server_and_command(server='remote', command='mymagma')   #␣
↪indirect doctest
No remote temporary directory (option server_tmpdir) specified, using /tmp/␣
↪on remote
>>> magma.server()
'remote'
>>> magma.command()
'ssh -t remote mymagma'
```

user_dir()

class sage.interfaces.expect.**ExpectElement**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: *InterfaceElement*, *ExpectElement*

Expect element.

class sage.interfaces.expect.**ExpectFunction**(*parent*, *name*)

Bases: *InterfaceFunction*

Expect function.

class sage.interfaces.expect.**FunctionElement**(*obj*, *name*)

Bases: *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.

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
```

(continues on next page)

(continued from previous page)

```
False
sage: gc.isenabled()
True
```

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

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

Return `True` if `x` is of type *ExpectElement*.

This function is deprecated; use `isinstance()` (of *sage.interfaces.abc.ExpectElement*) instead.

EXAMPLES:

```
sage: from sage.interfaces.expect import is_ExpectElement
sage: is_ExpectElement(2)
doctest:...: DeprecationWarning: the function is_ExpectElement is deprecated; use␣
↪isinstance(x, sage.interfaces.abc.ExpectElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
```

```
>>> from sage.all import *
>>> from sage.interfaces.expect import is_ExpectElement
>>> is_ExpectElement(Integer(2))
doctest:...: DeprecationWarning: the function is_ExpectElement is deprecated; use␣
↪isinstance(x, sage.interfaces.abc.ExpectElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
```

CHAPTER

THREE

SAGE WRAPPER AROUND PEXPECT'S `SPAWN` CLASS AND

the ptyprocess's `PtyProcess` class.

AUTHOR:

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

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

Bases: `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))
```

```
>>> from sage.all import *
>>> from sage.interfaces.sagespawn import SageSpawn
>>> s = SageSpawn("sleep 1000")
>>> s.close()
>>> while s.isalive():  # long time (5 seconds)
...     sleep(float(RealNumber('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"])
```

```
>>> from sage.all import *
>>> from sage.interfaces.sagespawn import SageSpawn
>>> 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:
....:         sleep(float(0.1))
....:     else:
....:         break  # process got killed
```

```
>>> from sage.all import *
>>> s.ptyproc.terminate_async(interval=float(RealNumber('0.2')))
>>> while True:
...     try:
...         os.kill(s.pid, Integer(0))
...     except OSError:
...         sleep(float(RealNumber('0.1')))
...     else:
...         break  # process got killed
```

class sage.interfaces.sagespawn.**SageSpawn**(**args*, ***kwds*)

Bases: `spawn`

Spawn a subprocess in a pseudo-tty.

- `*args`, `**kwds` – 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 ...
```

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

expect_peek(**args*, ***kwds*)

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"])
```

(continues on next page)

(continued from previous page)

```
sage: _ = E.expect_peek("w")
sage: E.read().decode('ascii')
'hello world\r\n'
```

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

expect_upto(**args*, ***kwds*)

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'
```

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

CHAPTER

FOUR

ABSTRACT BASE CLASSES FOR INTERFACE ELEMENTS

class sage.interfaces.abc.**AxiomElement**

Bases: object

Abstract base class for *AxiomElement*.

This class is defined for the purpose of isinstance tests. It should not be instantiated.

EXAMPLES:

By design, there is a unique direct subclass:

```
sage: len(sage.interfaces.abc.AxiomElement.__subclasses__()) <= 1
True
```

```
>>> from sage.all import *
>>> len(sage.interfaces.abc.AxiomElement.__subclasses__()) <= Integer(1)
True
```

class sage.interfaces.abc.**ExpectElement**

Bases: object

Abstract base class for *ExpectElement*.

This class is defined for the purpose of isinstance tests. It should not be instantiated.

EXAMPLES:

By design, there is a unique direct subclass:

```
sage: len(sage.interfaces.abc.ExpectElement.__subclasses__()) <= 1
True
```

```
>>> from sage.all import *
>>> len(sage.interfaces.abc.ExpectElement.__subclasses__()) <= Integer(1)
True
```

class sage.interfaces.abc.**FriCASElement**

Bases: object

Abstract base class for *FriCASElement*.

This class is defined for the purpose of isinstance tests. It should not be instantiated.

EXAMPLES:

By design, there is a unique direct subclass:

```
sage: len(sage.interfaces.abc.FriCASElement.__subclasses__()) <= 1
True
```

```
>>> from sage.all import *
>>> len(sage.interfaces.abc.FriCASElement.__subclasses__()) <= Integer(1)
True
```

class sage.interfaces.abc.**GapElement**

Bases: object

Abstract base class for *GapElement*.

This class is defined for the purpose of isinstance tests. It should not be instantiated.

EXAMPLES:

By design, there is a unique direct subclass:

```
sage: len(sage.interfaces.abc.GapElement.__subclasses__()) <= 1
True
```

```
>>> from sage.all import *
>>> len(sage.interfaces.abc.GapElement.__subclasses__()) <= Integer(1)
True
```

class sage.interfaces.abc.**GpElement**

Bases: object

Abstract base class for *GpElement*.

This class is defined for the purpose of isinstance tests. It should not be instantiated.

EXAMPLES:

By design, there is a unique direct subclass:

```
sage: len(sage.interfaces.abc.GpElement.__subclasses__()) <= 1
True
```

```
>>> from sage.all import *
>>> len(sage.interfaces.abc.GpElement.__subclasses__()) <= Integer(1)
True
```

class sage.interfaces.abc.**Macaulay2Element**

Bases: object

Abstract base class for *Macaulay2Element*.

This class is defined for the purpose of isinstance tests. It should not be instantiated.

EXAMPLES:

By design, there is a unique direct subclass:

```
sage: len(sage.interfaces.abc.Macaulay2Element.__subclasses__()) <= 1
True
```

```
>>> from sage.all import *
>>> len(sage.interfaces.abc.Macaulay2Element.__subclasses__()) <= Integer(1)
True
```

class sage.interfaces.abc.**MagmaElement**

Bases: `object`

Abstract base class for *MagmaElement*.

This class is defined for the purpose of `isinstance` tests. It should not be instantiated.

EXAMPLES:

By design, there is a unique direct subclass:

```
sage: len(sage.interfaces.abc.MagmaElement.__subclasses__()) <= 1
True
```

```
>>> from sage.all import *
>>> len(sage.interfaces.abc.MagmaElement.__subclasses__()) <= Integer(1)
True
```

class sage.interfaces.abc.**SingularElement**

Bases: `object`

Abstract base class for *SingularElement*.

This class is defined for the purpose of `isinstance` tests. It should not be instantiated.

EXAMPLES:

By design, there is a unique direct subclass:

```
sage: len(sage.interfaces.abc.SingularElement.__subclasses__()) <= 1
True
```

```
>>> from sage.all import *
>>> len(sage.interfaces.abc.SingularElement.__subclasses__()) <= Integer(1)
True
```

CHAPTER

FIVE

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 is 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
```

(continues on next page)

(continued from previous page)

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

```
>>> from sage.all import *
>>> axiom('3 * 5')                        #optional - axiom
15
>>> a = axiom(Integer(3)) * axiom(Integer(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
```

```
>>> from sage.all import *
>>> type(a)                               #optional - axiom
<class 'sage.interfaces.axiom.AxiomElement'>
>>> parent(a)                             #optional - axiom
Axiom
```

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

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

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

```
>>> from sage.all import *
>>> 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 )
>>> type(F)                                #optional - axiom
<class 'sage.interfaces.axiom.AxiomElement'>
>>> 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
```

```
>>> from sage.all import *
>>> a = axiom(Integer(2)/Integer(3)); a          #optional - axiom
  2
  -
  3
>>> 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
```

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

```
>>> from sage.all import *
>>> f = axiom('x^5 - y^5')                    #optional - axiom
>>> f**Integer(2)                                       #optional - axiom
   10      5 5    10
  y   - 2x y  + x
>>> 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.

```
sage:  f = axiom('(x^5 - y^5)^10000')       # not tested
Interrupting Axiom...
...
<class 'exceptions.TypeError'>: Ctrl-c pressed while running Axiom
```

```
>>> from sage.all import *
>>>  f = axiom('(x^5 - y^5)^10000')       # not tested
Interrupting Axiom...
...
<class 'exceptions.TypeError'>: Ctrl-c pressed while running Axiom
```

```
sage: axiom('1/100 + 1/101')                  #optional - axiom
   201
  -----
  10100
sage: a = axiom('(1 + sqrt(2))^5'); a         #optional - axiom
     +-+
  29\|2  + 41
```

```
>>> from sage.all import *
>>> axiom('1/100 + 1/101')                  #optional - axiom
   201
  -----
  10100
>>> a = axiom('(1 + sqrt(2))^5'); a         #optional - axiom
     +-+
  29\|2  + 41
```

class sage.interfaces.axiom.**Axiom**(*name='axiom'*, *command='axiom -nox -noclef'*, *script_subdirectory=None*, *logfile=None*, *server=None*, *server_tmpdir=None*, *init_code=[')lisp (si::readline-off)']*)

Bases: *PanAxiom*

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.
-----------------------------------------------------------------------------
```

```
>>> from sage.all import *
>>> axiom.console() #not tested
```

(continues on next page)

(continued from previous page)

```
                    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.**AxiomElement**

alias of *PanAxiomElement*

sage.interfaces.axiom.**AxiomExpectFunction**

alias of *PanAxiomExpectFunction*

sage.interfaces.axiom.**AxiomFunctionElement**

alias of *PanAxiomFunctionElement*

class sage.interfaces.axiom.**PanAxiom**(*name='axiom'*, *command='axiom -nox -noclef'*, *script_subdirectory=None*, *logfile=None*, *server=None*, *server_tmpdir=None*, *init_code=[')lisp (si::readline-off)']*)

Bases: ExtraTabCompletion, *Expect*

Interface to a PanAxiom interpreter.

get(*var*)

Get the string value of the Axiom variable var.

EXAMPLES:

```
sage: # optional - axiom
sage: axiom.set('xx', '2')
sage: axiom.get('xx')
'2'
sage: a = axiom('(1 + sqrt(2))^5')
sage: axiom.get(a.name())
'     +-+\r\r\n  29\\|2  + 41'
```

```
>>> from sage.all import *
>>> # optional - axiom
>>> axiom.set('xx', '2')
>>> axiom.get('xx')
'2'
>>> a = axiom('(1 + sqrt(2))^5')
>>> axiom.get(a.name())
'     +-+\r\r\n  29\\|2  + 41'
```

set(*var*, *value*)

Set the variable var to the given value.

EXAMPLES:

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

```
>>> from sage.all import *
>>> axiom.set('xx', '2')     #optional - axiom
>>> axiom.get('xx')          #optional - axiom
'2'
```

class sage.interfaces.axiom.**PanAxiomElement**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: `ExpectElement`, `AxiomElement`

as_type(*type*)

Return `self` as type.

EXAMPLES:

```
sage: a = axiom(1.2); a            #optional - axiom
1.2
sage: a.as_type(axiom.DoubleFloat) #optional - axiom
1.2
sage: _.type()                     #optional - axiom
DoubleFloat
```

```
>>> from sage.all import *
>>> a = axiom(RealNumber('1.2')); a            #optional - axiom
1.2
>>> a.as_type(axiom.DoubleFloat) #optional - axiom
1.2
>>> _.type()                     #optional - axiom
DoubleFloat
```

comma(**args*)

Return an Axiom tuple from `self` and `args`.

EXAMPLES:

```
sage: # optional - axiom
sage: two = axiom(2)
sage: two.comma(3)
[2,3]
sage: two.comma(3,4)
[2,3,4]
sage: _.type()
Tuple PositiveInteger
```

```
>>> from sage.all import *
>>> # optional - axiom
>>> two = axiom(Integer(2))
>>> two.comma(Integer(3))
[2,3]
>>> two.comma(Integer(3),Integer(4))
[2,3,4]
>>> _.type()
Tuple PositiveInteger
```

type()

Return the type of an AxiomElement.

EXAMPLES:

```
sage: axiom(x+2).type()   #optional - axiom
Polynomial Integer
```

```
>>> from sage.all import *
>>> axiom(x+Integer(2)).type()   #optional - axiom
Polynomial Integer
```

unparsed_input_form()

Get the linear string representation of this object, if possible (often it isn't).

EXAMPLES:

```
sage: a = axiom(x^2+1); a     #optional - axiom
   2
  x  + 1
sage: a.unparsed_input_form() #optional - axiom
'x*x+1'
```

```
>>> from sage.all import *
>>> a = axiom(x**Integer(2)+Integer(1)); a     #optional - axiom
   2
  x  + 1
>>> a.unparsed_input_form() #optional - axiom
'x*x+1'
```

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

Bases: *ExpectFunction*

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

Bases: *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.
-----------------------------------------------------------------------------
```

```
>>> from sage.all import *
>>> axiom_console() #not tested
                        AXIOM Computer Algebra System
                        Version: Axiom (January 2009)
               Timestamp: Sunday January 25, 2009 at 07:08:54
-----------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
   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*)

Return True if x is of type *AxiomElement*.

EXAMPLES:

```
sage: from sage.interfaces.axiom import is_AxiomElement
sage: is_AxiomElement(2)
doctest:...: DeprecationWarning: the function is_AxiomElement is deprecated; use␣
↪isinstance(x, sage.interfaces.abc.AxiomElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
sage: is_AxiomElement(axiom(2))  # optional - axiom
True
```

```
>>> from sage.all import *
>>> from sage.interfaces.axiom import is_AxiomElement
>>> is_AxiomElement(Integer(2))
doctest:...: DeprecationWarning: the function is_AxiomElement is deprecated; use␣
↪isinstance(x, sage.interfaces.abc.AxiomElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
>>> is_AxiomElement(axiom(Integer(2)))  # optional - axiom
True
```

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

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

EXAMPLES:

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

```
>>> from sage.all import *
>>> from sage.interfaces.axiom import reduce_load_Axiom
>>> reduce_load_Axiom()
Axiom
```

CHAPTER

SIX

THE ELLIPTIC CURVE FACTORIZATION METHOD

The elliptic curve factorization method (ECM) is the fastest way to factor a **known composite** integer if one of the factors is relatively small (up to approximately 80 bits / 25 decimal digits). To factor an arbitrary integer it must be combined with a primality test. The `ECM.factor()` method is an example for how to combine ECM with a primality test to compute the prime factorization of integers.

Sage includes GMP-ECM, which is a highly optimized implementation of Lenstra's elliptic curve factorization method. See https://gitlab.inria.fr/zimmerma/ecm for more about GMP-ECM.

AUTHORS:

These people wrote GMP-ECM: Pierrick Gaudry, Jim Fougeron, Laurent Fousse, Alexander Kruppa, Dave Newman, Paul Zimmermann

BUGS:

Output from ecm is non-deterministic. Doctests should set the random seed, but currently there is no facility to do so.

class sage.interfaces.ecm.**ECM**(*B1=10*, *B2=None*, ***kwds*)

Bases: `SageObject`

Create an interface to the GMP-ECM elliptic curve method factorization program.

See https://gitlab.inria.fr/zimmerma/ecm

INPUT:

- `B1` – integer; Stage 1 bound
- `B2` – integer; Stage 2 bound (or interval B2min-B2max)

In addition the following keyword arguments can be used:

- `x0` – integer x; use x as initial point
- `sigma` – integer s; use s as curve generator [ecm]
- `A` – integer a; use a as curve parameter [ecm]
- `k` – integer n; perform $>= n$ steps in stage 2
- `power` – integer n; use x^n for Brent-Suyama's extension
- `dickson` – integer n; use n-th Dickson's polynomial for Brent-Suyama's extension
- `c` – integer n; perform n runs for each input
- `pm1` – boolean; perform P-1 instead of ECM
- `pp1` – boolean; perform P+1 instead of ECM
- `q` – boolean; quiet mode

- `v` – boolean; verbose mode
- `timestamp` – boolean; print a time stamp with each number
- `mpzmod` – boolean; use GMP's mpz_mod for mod reduction
- `modmuln` – boolean; use Montgomery's MODMULN for mod reduction
- `redc` – boolean; use Montgomery's REDC for mod reduction
- `nobase2` – boolean; disable special base-2 code
- `base2` – integer n; force base 2 mode with 2^n+1 (n>0) or 2^n-1 (n<0)
- `save` – string filename; save residues at end of stage 1 to file
- `savea` – string filename; Like -save, appends to existing files
- `resume` – string filename; resume residues from file, reads from stdin if file is "-"
- `primetest` – boolean; perform a primality test on input
- `treefile` – string; store product tree of F in files f.0 f.1 …
- `i` – integer; increment B1 by this constant on each run
- `I` – integer f; auto-calculated increment for B1 multiplied by f scale factor
- `inp` – string; use file as input (instead of redirecting stdin)
- `b` – boolean; use breadth-first mode of file processing
- `d` – boolean; use depth-first mode of file processing (default)
- `one` – boolean; stop processing a candidate if a factor is found (looping mode)
- `n` – boolean; run ecm in 'nice' mode (below normal priority)
- `nn` – boolean; run ecm in 'very nice' mode (idle priority)
- `t` – integer n; trial divide candidates before P-1, P+1 or ECM up to n
- `ve` – integer n; verbosely show short ($< n$ character) expressions on each loop
- `B2scale` – integer; multiplies the default B2 value
- `go` – integer; preload with group order val, which can be a simple expression, or can use N as a placeholder for the number being factored
- `prp` – string; use shell command cmd to do large primality tests
- `prplen` – integer; only candidates longer than this number of digits are 'large'
- `prpval` – integer; value>=0 which indicates the prp command foundnumber to be PRP
- `prptmp` – file; outputs n value to temp file prior to running (NB. gets deleted)
- `prplog` – file; otherwise get PRP results from this file (NB. gets deleted)
- `prpyes` – string; literal string found in prplog file when number is PRP
- `prpno` – string; literal string found in prplog file when number is composite

factor(*n*, *factor_digits=None*, *B1=2000*, *proof=False*, ***kwds*)

Return a probable prime factorization of n.

Combines GMP-ECM with a primality test, see `is_prime()`. The primality test is provable or probabilistic depending on the $proof$ flag.

Moreover, for small n PARI is used directly.

> **Warning**
>
> There is no mathematical guarantee that the factors returned are actually prime if `proof=False` (default). It is extremely likely, though. Currently, there are no known examples where this fails.

INPUT:

- `n` – positive integer
- `factor_digits` – integer or `None` (default); optional guess at how many digits are in the smallest factor
- `B1` – initial lower bound, defaults to 2000 (15 digit factors); used if `factor_digits` is not specified
- `proof` – boolean (default: `False`); whether to prove that the factors are prime
- `kwds` – keyword arguments to pass to ecm-gmp; see help for `ECM` for more details

OUTPUT: list of integers whose product is n

> **Note**
>
> Trial division should typically be performed, but this is not implemented (yet) in this method.
>
> If you suspect that n is the product of two similarly-sized primes, other methods (such as a quadratic sieve – use the qsieve command) will usually be faster.
>
> The best known algorithm for factoring in the case where all factors are large is the general number field sieve. This is not implemented in Sage; You probably want to use a cluster for problems of this size.

EXAMPLES:

```
sage: ecm.factor(602400691612422154516282778947806249229526581)
[45949729863572179, 13109994191499930367061460439]
sage: ecm.factor((2^197 + 1)/3)  # long time
[197002597249, 1348959352853811313, 251951573867253012259144010843]
sage: ecm.factor(179427217^13) == [179427217] * 13
True
```

```
>>> from sage.all import *
>>> ecm.factor(Integer(602400691612422154516282778947806249229526581))
[45949729863572179, 13109994191499930367061460439]
>>> ecm.factor((Integer(2)**Integer(197) + Integer(1))/Integer(3))  # long␣
↪time
[197002597249, 1348959352853811313, 251951573867253012259144010843]
>>> ecm.factor(Integer(179427217)**Integer(13)) == [Integer(179427217)] *␣
↪Integer(13)
True
```

find_factor(*n*, *factor_digits=None*, *B1=2000*, ***kwds*)

Return a factor of n.

See also `factor()` if you want a prime factorization of n.

INPUT:

- `n` – positive integer
- `factor_digits` – integer or `None` (default); decimal digits estimate of the wanted factor

- `B1` – integer; Stage 1 bound (default: 2000). This is used as bound if `factor_digits` is not specified
- `kwds` – optional keyword parameters

OUTPUT:

List of integers whose product is n. For certain lengths of the factor, this is the best algorithm to find a factor.

> **Note**
>
> ECM is not a good primality test. Not finding a factorization is only weak evidence for n being prime. You should run a **good** primality test before calling this function.

EXAMPLES:

```
sage: f = ECM()
sage: n = 508021860739623467191080372196682785441177798407961
sage: f.find_factor(n)
[79792266297612017, 6366805760909027985741435139224233]
```

```
>>> from sage.all import *
>>> f = ECM()
>>> n = Integer(508021860739623467191080372196682785441177798407961)
>>> f.find_factor(n)
[79792266297612017, 6366805760909027985741435139224233]
```

Note that the input number cannot have more than 4095 digits:

```
sage: f = 2^2^14+1
sage: ecm.find_factor(f)
Traceback (most recent call last):
...
ValueError: n must have at most 4095 digits
```

```
>>> from sage.all import *
>>> f = Integer(2)**Integer(2)**Integer(14)+Integer(1)
>>> ecm.find_factor(f)
Traceback (most recent call last):
...
ValueError: n must have at most 4095 digits
```

get_last_params()

Return the parameters (including the curve) of the last ecm run.

In the case that the number was factored successfully, this will return the parameters that yielded the factorization.

OUTPUT:

A dictionary containing the parameters for the most recent factorization.

EXAMPLES:

```
sage: ecm.factor((2^197 + 1)/3)              # long time
[197002597249, 1348959352853811313, 251951573867253012259144010843]
```

(continues on next page)

(continued from previous page)

```
sage: ecm.get_last_params()                    # random output
{'poly': 'x^1', 'sigma': '1785694449', 'B1': '8885', 'B2': '1002846'}
```

```
>>> from sage.all import *
>>> ecm.factor((Integer(2)**Integer(197) + Integer(1))/Integer(3))               ␣
↪ # long time
[197002597249, 1348959352853811313, 251951573867253012259144010843]
>>> ecm.get_last_params()                    # random output
{'poly': 'x^1', 'sigma': '1785694449', 'B1': '8885', 'B2': '1002846'}
```

interact()

Interactively interact with the ECM program.

EXAMPLES:

```
sage: ecm.interact()     # not tested
```

```
>>> from sage.all import *
>>> ecm.interact()     # not tested
```

one_curve(*n*, *factor_digits=None*, *B1=2000*, *algorithm='ECM'*, ***kwds*)

Run one single ECM (or P-1/P+1) curve on input n.

Note that trying a single curve is not particularly useful by itself. One typically needs to run over thousands of trial curves to factor n.

INPUT:

- `n` – positive integer
- `factor_digits` – integer; decimal digits estimate of the wanted factor
- `B1` – integer; Stage 1 bound (default: 2000)
- `algorithm` – either “ECM” (default); “P-1” or “P+1”

OUTPUT:

A list `[p, q]` where p and q are integers and n = p * q. If no factor was found, then p = 1 and q = n.

> **Warning**
>
> Neither p nor q in the output is guaranteed to be prime.

EXAMPLES:

```
sage: f = ECM()
sage: n = 508021860739623467191080372196682785441177798407961
sage: f.one_curve(n, B1=10000, sigma=11)
[1, 508021860739623467191080372196682785441177798407961]
sage: f.one_curve(n, B1=10000, sigma=1022170541)
[79792266297612017, 6366805760909027985741435139224233]
sage: n = 432132887883903108009802143314445113500016816977037257
sage: f.one_curve(n, B1=500000, algorithm='P-1')
[67872792749091946529, 6366805760909027985741435139224233]
```

(continues on next page)

(continued from previous page)

```
sage: n = 2088352670731726262548647919416588631875815083
sage: f.one_curve(n, B1=2000, algorithm='P+1', x0=5)
[328006342451, 6366805760909027985741435139224233]
```

```
>>> from sage.all import *
>>> f = ECM()
>>> n = Integer(508021860739623467191080372196682785441177798407961)
>>> f.one_curve(n, B1=Integer(10000), sigma=Integer(11))
[1, 508021860739623467191080372196682785441177798407961]
>>> f.one_curve(n, B1=Integer(10000), sigma=Integer(1022170541))
[79792266297612017, 6366805760909027985741435139224233]
>>> n = Integer(432132887883903108009802143314445113500016816977037257)
>>> f.one_curve(n, B1=Integer(500000), algorithm='P-1')
[67872792749091946529, 6366805760909027985741435139224233]
>>> n = Integer(2088352670731726262548647919416588631875815083)
>>> f.one_curve(n, B1=Integer(2000), algorithm='P+1', x0=Integer(5))
[328006342451, 6366805760909027985741435139224233]
```

recommended_B1 (*factor_digits*)

Return recommended `B1` setting.

INPUT:

- `factor_digits` – integer; number of digits

OUTPUT:

Integer. Recommended settings from http://www.mersennewiki.org/index.php/Elliptic_Curve_Method

EXAMPLES:

```
sage: ecm.recommended_B1(33)
1000000
```

```
>>> from sage.all import *
>>> ecm.recommended_B1(Integer(33))
1000000
```

time (*n*, *factor_digits*, *verbose=False*)

Print a runtime estimate.

BUGS:

This method should really return something and not just print stuff on the screen.

INPUT:

- `n` – positive integer
- `factor_digits` – the (estimated) number of digits of the smallest factor

OUTPUT:

An approximation for the amount of time it will take to find a factor of size factor_digits in a single process on the current computer. This estimate is provided by GMP-ECM's verbose option on a single run of a curve.

EXAMPLES:

```
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, sqrredc: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
```

```
>>> from sage.all import *
>>> n = next_prime(Integer(11)**Integer(23))*next_
↪prime(Integer(11)**Integer(37))
>>> ecm.time(n, Integer(35))                    # random output
Expected curves: 910, Expected time: 23.95m

>>> ecm.time(n, Integer(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␣
```

(continues on next page)

(continued from previous page)

```
↪304481639541418099574459496544854621998616257489887231115912293 (63 digits)
Using MODMULN [mulredc:0, sqrredc: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
<BLANKLINE>
Expected curves: 4911, Expected time: 32.25m
```

CHAPTER SEVEN

INTERFACE TO 4TI2

https://4ti2.github.io/

You must have the 4ti2 Sage package installed on your computer for this interface to work.

Use `sage -i 4ti2` to install the package.

AUTHORS:

- Mike Hansen (2009): Initial version.
- Bjarke Hammersholt Roune (2009-06-26): Added Groebner, made code usable as part of the Sage library and added documentation and some doctests.
- Marshall Hampton (2011): Minor fixes to documentation.

class `sage.interfaces.four_ti_2.`**`FourTi2`**(*directory=None*)

Bases: `object`

An interface to the program 4ti2.

Each 4ti2 command is exposed as a method of this class.

`call`(*command*, *project*, *verbose*, *options=True*)

Run the 4ti2 program `command` on the project named `project` in the directory `directory()`.

INPUT:

- `command` – the 4ti2 program to run
- `project` – the file name of the project to run on
- `verbose` – display the output of 4ti2 if `True`
- `options` – list of strings to pass to the program

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: four_ti_2.write_matrix([[6,10,15]], "test_file")
sage: four_ti_2.call("groebner", "test_file", False)  # optional - 4ti2
sage: four_ti_2.read_matrix("test_file.gro")  # optional - 4ti2
[-5  0  2]
[-5  3  0]
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> four_ti_2.write_matrix([[Integer(6),Integer(10),Integer(15)]], "test_file
↪")
```

(continues on next page)

(continued from previous page)

```
>>> four_ti_2.call("groebner", "test_file", False)  # optional - 4ti2
>>> four_ti_2.read_matrix("test_file.gro")  # optional - 4ti2
[-5  0  2]
[-5  3  0]
```

circuits(*mat=None*, *project=None*)

Run the 4ti2 program `circuits` on the parameters.

See 4ti2 website for details.

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: four_ti_2.circuits([1,2,3])  # optional - 4ti2
[ 0  3 -2]
[ 2 -1  0]
[ 3  0 -1]
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> four_ti_2.circuits([Integer(1),Integer(2),Integer(3)])  # optional - 4ti2
[ 0  3 -2]
[ 2 -1  0]
[ 3  0 -1]
```

directory()

Return the directory where the input files for 4ti2 are written by Sage and where 4ti2 is run.

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import FourTi2
sage: f = FourTi2("/tmp/")
sage: f.directory()
'/tmp/'
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import FourTi2
>>> f = FourTi2("/tmp/")
>>> f.directory()
'/tmp/'
```

graver(*mat=None*, *lat=None*, *project=None*)

Run the 4ti2 program `graver` on the parameters.

See 4ti2 website for details.

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: four_ti_2.graver([1,2,3])  # optional - 4ti2
[ 2 -1  0]
[ 3  0 -1]
[ 1  1 -1]
[ 1 -2  1]
```

(continues on next page)

(continued from previous page)

```
[ 0  3 -2]
sage: four_ti_2.graver(lat=[[1,2,3],[1,1,1]])  # optional - 4ti2
[ 1  0 -1]
[ 0  1  2]
[ 1  1  1]
[ 2  1  0]
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> four_ti_2.graver([Integer(1),Integer(2),Integer(3)])  # optional - 4ti2
[ 2 -1  0]
[ 3  0 -1]
[ 1  1 -1]
[ 1 -2  1]
[ 0  3 -2]
>>> four_ti_2.graver(lat=[[Integer(1),Integer(2),Integer(3)],[Integer(1),
↪Integer(1),Integer(1)]])  # optional - 4ti2
[ 1  0 -1]
[ 0  1  2]
[ 1  1  1]
[ 2  1  0]
```

groebner(*mat=None*, *lat=None*, *project=None*)

Run the 4ti2 program `groebner` on the parameters.

This computes a toric Groebner basis of a matrix.

See 4ti2 website for details.

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: A = [6,10,15]
sage: four_ti_2.groebner(A)  # optional - 4ti2
[-5  0  2]
[-5  3  0]
sage: four_ti_2.groebner(lat=[[1,2,3],[1,1,1]])  # optional - 4ti2
[-1  0  1]
[ 2  1  0]
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> A = [Integer(6),Integer(10),Integer(15)]
>>> four_ti_2.groebner(A)  # optional - 4ti2
[-5  0  2]
[-5  3  0]
>>> four_ti_2.groebner(lat=[[Integer(1),Integer(2),Integer(3)],[Integer(1),
↪Integer(1),Integer(1)]])  # optional - 4ti2
[-1  0  1]
[ 2  1  0]
```

hilbert(*mat=None*, *lat=None*, *project=None*)

Run the 4ti2 program `hilbert` on the parameters.

See 4ti2 website for details.

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: four_ti_2.hilbert(four_ti_2._magic3x3())   # optional - 4ti2
[2 0 1 0 1 2 1 2 0]
[1 0 2 2 1 0 0 2 1]
[0 2 1 2 1 0 1 0 2]
[1 2 0 0 1 2 2 0 1]
[1 1 1 1 1 1 1 1 1]
sage: four_ti_2.hilbert(lat=[[1,2,3],[1,1,1]])   # optional - 4ti2
[2 1 0]
[0 1 2]
[1 1 1]
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> four_ti_2.hilbert(four_ti_2._magic3x3())   # optional - 4ti2
[2 0 1 0 1 2 1 2 0]
[1 0 2 2 1 0 0 2 1]
[0 2 1 2 1 0 1 0 2]
[1 2 0 0 1 2 2 0 1]
[1 1 1 1 1 1 1 1 1]
>>> four_ti_2.hilbert(lat=[[Integer(1),Integer(2),Integer(3)],[Integer(1),
↪Integer(1),Integer(1)]])   # optional - 4ti2
[2 1 0]
[0 1 2]
[1 1 1]
```

minimize(*mat=None*, *lat=None*)

Run the 4ti2 program `minimize` on the parameters.

See 4ti2 website for details.

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: four_ti_2.minimize()   # optional - 4ti2
Traceback (most recent call last):
...
NotImplementedError: 4ti2 command 'minimize' not implemented in Sage.
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> four_ti_2.minimize()   # optional - 4ti2
Traceback (most recent call last):
...
NotImplementedError: 4ti2 command 'minimize' not implemented in Sage.
```

ppi(*n*)

Run the 4ti2 program `ppi` on the parameters.

See 4ti2 website for details.

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: four_ti_2.ppi(3)  # optional - 4ti2
[-2  1  0]
[ 0 -3  2]
[-1 -1  1]
[-3  0  1]
[ 1 -2  1]
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> four_ti_2.ppi(Integer(3))  # optional - 4ti2
[-2  1  0]
[ 0 -3  2]
[-1 -1  1]
[-3  0  1]
[ 1 -2  1]
```

qsolve(*mat=None*, *rel=None*, *sign=None*, *project=None*)

Run the 4ti2 program `qsolve` on the parameters.

See 4ti2 website for details.

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: A = [[1,1,1],[1,2,3]]
sage: four_ti_2.qsolve(A)  # optional - 4ti2
[[], [ 1 -2  1]]
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> A = [[Integer(1),Integer(1),Integer(1)],[Integer(1),Integer(2),
↪Integer(3)]]
>>> four_ti_2.qsolve(A)  # optional - 4ti2
[[], [ 1 -2  1]]
```

rays(*mat=None*, *project=None*)

Run the 4ti2 program `rays` on the parameters.

See 4ti2 website for details.

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: four_ti_2.rays(four_ti_2._magic3x3())  # optional - 4ti2
[0 2 1 2 1 0 1 0 2]
[1 0 2 2 1 0 0 2 1]
[1 2 0 0 1 2 2 0 1]
[2 0 1 0 1 2 1 2 0]
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> four_ti_2.rays(four_ti_2._magic3x3())  # optional - 4ti2
[0 2 1 2 1 0 1 0 2]
```

(continues on next page)

(continued from previous page)

```
[1 0 2 2 1 0 0 2 1]
[1 2 0 0 1 2 2 0 1]
[2 0 1 0 1 2 1 2 0]
```

read_matrix(*filename*)

Read a matrix in 4ti2 format from the file `filename` in directory `directory()`.

INPUT:

- `filename` – the name of the file to read from

OUTPUT: the data from the file as a matrix over $\mathbf{Z}$

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: four_ti_2.write_matrix([[1,2,3],[3,4,6]], "test_file")
sage: four_ti_2.read_matrix("test_file")
[1 2 3]
[3 4 6]
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> four_ti_2.write_matrix([[Integer(1),Integer(2),Integer(3)],[Integer(3),
↪Integer(4),Integer(6)]], "test_file")
>>> four_ti_2.read_matrix("test_file")
[1 2 3]
[3 4 6]
```

temp_project()

Return an input project file name that has not been used yet.

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: four_ti_2.temp_project()
'project_...'
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> four_ti_2.temp_project()
'project_...'
```

write_array(*array*, *nrows*, *ncols*, *filename*)

Write the integer matrix `array` to the file `filename` in directory `directory()` in 4ti2 format.

The matrix must have `nrows` rows and `ncols` columns. It can be provided as a list of lists.

INPUT:

- `array` – a matrix of integers. Can be represented as a list of lists
- `nrows` – the number of rows in `array`
- `ncols` – the number of columns in `array`
- `file` – a file name not including a path

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: four_ti_2.write_array([[1,2,3],[3,4,5]], 2, 3, "test_file")
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> four_ti_2.write_array([[Integer(1),Integer(2),Integer(3)],[Integer(3),
↪Integer(4),Integer(5)]], Integer(2), Integer(3), "test_file")
```

write_matrix(*mat*, *filename*)

Write the matrix `mat` to the file `filename` in 4ti2 format.

INPUT:

- `mat` – a matrix of integers or something that can be converted to that
- `filename` – a file name not including a path

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: four_ti_2.write_matrix([[1,2],[3,4]], "test_file")
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> four_ti_2.write_matrix([[Integer(1),Integer(2)],[Integer(3),Integer(4)]],
↪"test_file")
```

write_single_row(*row*, *filename*)

Write the list `row` to the file `filename` in 4ti2 format as a matrix with one row.

INPUT:

- `row` – list of integers
- `filename` – a file name not including a path

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: four_ti_2.write_single_row([1,2,3,4], "test_file")
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> four_ti_2.write_single_row([Integer(1),Integer(2),Integer(3),Integer(4)],
↪"test_file")
```

zsolve(*mat=None*, *rel=None*, *rhs=None*, *sign=None*, *lat=None*, *project=None*)

Run the 4ti2 program `zsolve` on the parameters.

See 4ti2 website for details.

EXAMPLES:

```
sage: from sage.interfaces.four_ti_2 import four_ti_2
sage: A = [[1,1,1],[1,2,3]]
```

(continues on next page)

(continued from previous page)

```
sage: rel = ['<', '<']
sage: rhs = [2, 3]
sage: sign = [1,0,1]
sage: four_ti_2.zsolve(A, rel, rhs, sign)  # optional - 4ti2
[
         [ 1 -1  0]
         [ 0 -1  0]
[0 0 1]  [ 0 -3  2]
[1 1 0]  [ 1 -2  1]
[0 1 0], [ 0 -2  1], []
]
sage: four_ti_2.zsolve(lat=[[1,2,3],[1,1,1]])  # optional - 4ti2
[
             [1 2 3]
[0 0 0], [], [1 1 1]
]
```

```
>>> from sage.all import *
>>> from sage.interfaces.four_ti_2 import four_ti_2
>>> A = [[Integer(1),Integer(1),Integer(1)],[Integer(1),Integer(2),
↪Integer(3)]]
>>> rel = ['<', '<']
>>> rhs = [Integer(2), Integer(3)]
>>> sign = [Integer(1),Integer(0),Integer(1)]
>>> four_ti_2.zsolve(A, rel, rhs, sign)  # optional - 4ti2
[
         [ 1 -1  0]
         [ 0 -1  0]
[0 0 1]  [ 0 -3  2]
[1 1 0]  [ 1 -2  1]
[0 1 0], [ 0 -2  1], []
]
>>> four_ti_2.zsolve(lat=[[Integer(1),Integer(2),Integer(3)],[Integer(1),
↪Integer(1),Integer(1)]])  # optional - 4ti2
[
             [1 2 3]
[0 0 0], [], [1 1 1]
]
```

CHAPTER

EIGHT

INTERFACE TO FRICAS

Todo

- some conversions in `sage.functions` are still missing and all should be checked and tested

FriCAS is a free GPL-compatible (modified BSD license) general purpose computer algebra system based on Axiom. The FriCAS website can be found at http://fricas.sourceforge.net/.

AUTHORS:

- Mike Hansen (2009-02): Split off the FriCAS interface from the Axiom interface.
- Martin Rubey, Bill Page (2016-08): Completely separate from Axiom, implement more complete translation from FriCAS to SageMath types.

EXAMPLES:

```
sage: fricas('3 * 5')
15
sage: a = fricas(3) * fricas(5); a
15
```

```
>>> from sage.all import *
>>> fricas('3 * 5')
15
>>> a = fricas(Integer(3)) * fricas(Integer(5)); a
15
```

The type of a is *FriCASElement*, i.e., an element of the FriCAS interpreter:

```
sage: type(a)
<class 'sage.interfaces.fricas.FriCASElement'>
sage: a.parent()
FriCAS
```

```
>>> from sage.all import *
>>> type(a)
<class 'sage.interfaces.fricas.FriCASElement'>
>>> a.parent()
FriCAS
```

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

```
sage: a.typeOf()
PositiveInteger
```

```
>>> from sage.all import *
>>> a.typeOf()
PositiveInteger
```

FriCAS objects are normally displayed using "ASCII art":

```
sage: fricas(2/3)
  2
  -
  3
sage: fricas('x^2 + 3/7')
   2   3
  x  + -
       7
```

```
>>> from sage.all import *
>>> fricas(Integer(2)/Integer(3))
  2
  -
  3
>>> fricas('x^2 + 3/7')
   2   3
  x  + -
       7
```

Functions defined in FriCAS are available as methods of the *fricas* object:

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

```
>>> from sage.all import *
>>> F = fricas.factor('x^5 - y^5'); F
           4      3     2 2     3     4
- (y - x)(y  + x y  + x y  + x y + x )
>>> type(F)
<class 'sage.interfaces.fricas.FriCASElement'>
>>> F.typeOf()
Factored(Polynomial(Integer))
```

We can also create a FriCAS polynomial and apply the function `factor` from FriCAS. The notation `f.factor()` is consistent with how the rest of SageMath works:

```
sage: f = fricas('x^5 - y^5')
sage: f^2
```

(continues on next page)

(continued from previous page)

```
 10       5 5     10
y    - 2 x y   + x
sage: f.factor()
             4       3     2 2     3      4
- (y - x)(y   + x y   + x y   + x y + x )
```

```
>>> from sage.all import *
>>> f = fricas('x^5 - y^5')
>>> f**Integer(2)
 10       5 5     10
y    - 2 x y   + x
>>> f.factor()
             4       3     2 2     3      4
- (y - x)(y   + x y   + x y   + x y + x )
```

For many FriCAS types, translation to an appropriate SageMath type is available:

```
sage: f.factor().sage()
(y - x) * (y^4 + y^3*x + y^2*x^2 + y*x^3 + x^4)
```

```
>>> from sage.all import *
>>> f.factor().sage()
(y - x) * (y^4 + y^3*x + y^2*x^2 + y*x^3 + x^4)
```

Control-C interruption works well with the FriCAS interface. For example, try the following sum but with a much bigger range, and hit control-C:

```
sage:  f = fricas('(x^5 - y^5)^10000')                                    # not␣
↪tested
Interrupting FriCAS...
...
KeyboardInterrupt: Ctrl-c pressed while running FriCAS
```

```
>>>  f = fricas('(x^5 - y^5)^10000')                                    # not tested
Interrupting FriCAS...
...
KeyboardInterrupt: Ctrl-c pressed while running FriCAS
```

Let us demonstrate some features of FriCAS. FriCAS can guess a differential equation for the generating function for integer partitions:

```
sage: fricas("guessADE([partition n for n in 0..40], homogeneous==4)")
[
  [
      n
    [x ]f(x):
             2     3 (iv)            2     2 ,                 3   ,,,
           x f(x) f     (x) + (20 x f(x) f (x) + 5 x f(x) )f    (x)


        +
                  2     2 ,,    2
```

(continues on next page)

(continued from previous page)

```
          - 39 x f(x) f   (x)

        +
                 2      ,   2             2 ,              3   ,,           2 ,    4
          (12 x f(x)f  (x)   - 15 x f(x) f  (x) + 4 f(x)  )f   (x) + 6 x f (x)

        +
                     ,   3          2 ,    2
          10 x f(x)f  (x)   - 16 f(x)  f  (x)

      =
        0
    ,
                       2      3      4
     f(x) = 1 + x + 2 x   + 3 x   + O(x )]
   ]
```

```
>>> from sage.all import *
>>> fricas("guessADE([partition n for n in 0..40], homogeneous==4)")
[
  [
      n
    [x ]f(x):
           2    3 (iv)            2    2 ,               3   ,,,
          x f(x) f     (x) + (20 x f(x) f  (x) + 5 x f(x) )f    (x)
<BLANKLINE>
        +
                  2    2 ,,   2
          - 39 x f(x) f   (x)
<BLANKLINE>
        +
                 2      ,   2             2 ,              3   ,,           2 ,    4
          (12 x f(x)f  (x)   - 15 x f(x) f  (x) + 4 f(x)  )f   (x) + 6 x f (x)
<BLANKLINE>
        +
                     ,   3          2 ,    2
          10 x f(x)f  (x)   - 16 f(x)  f  (x)
<BLANKLINE>
      =
        0
    ,
                       2      3      4
     f(x) = 1 + x + 2 x   + 3 x   + O(x )]
   ]
```

FriCAS can solve linear ordinary differential equations:

```
sage: fricas.set("y", "operator y")
sage: fricas.set("deq", "x^3*D(y x, x, 3) + x^2*D(y x, x, 2) - 2*x*D(y x, x) + 2*y x -
↪ 2*x^4")
sage: fricas.set("sol", "solve(deq, y, x)"); fricas("sol")
               5       3       2
```

(continues on next page)

(continued from previous page)

```
                x   - 10 x   + 20 x   + 4
[particular = -----------------------,
                          15 x
              3       2        3        3       2
           2 x   - 3 x   + 1   x   - 1   x   - 3 x   - 1
 basis = [---------------, ------, -------------]]
                  x             x           x

sage: fricas("sol.particular").sage()
1/15*(x^5 - 10*x^3 + 20*x^2 + 4)/x
sage: fricas("sol.basis").sage()
[(2*x^3 - 3*x^2 + 1)/x, (x^3 - 1)/x, (x^3 - 3*x^2 - 1)/x]
sage: fricas.eval(")clear values y deq sol")
''
```

```
>>> from sage.all import *
>>> fricas.set("y", "operator y")
>>> fricas.set("deq", "x^3*D(y x, x, 3) + x^2*D(y x, x, 2) - 2*x*D(y x, x) + 2*y x -␣
↪2*x^4")
>>> fricas.set("sol", "solve(deq, y, x)"); fricas("sol")
                  5        3        2
                x   - 10 x   + 20 x   + 4
[particular = -----------------------,
                          15 x
              3       2        3        3       2
           2 x   - 3 x   + 1   x   - 1   x   - 3 x   - 1
 basis = [---------------, ------, -------------]]
                  x             x           x

>>> fricas("sol.particular").sage()
1/15*(x^5 - 10*x^3 + 20*x^2 + 4)/x
>>> fricas("sol.basis").sage()
[(2*x^3 - 3*x^2 + 1)/x, (x^3 - 1)/x, (x^3 - 3*x^2 - 1)/x]
>>> fricas.eval(")clear values y deq sol")
''
```

FriCAS can expand expressions into series:

```
sage: x = var('x'); ex = sqrt(cos(x)); a = fricas(ex).series(x=0); a
    1  2     1  4     19   6      559   8      29161    10       11
1 - - x  - -- x  - ---- x  - ------ x  - --------- x   + O(x   )
    4      96      5760      645120      116121600

sage: a.coefficients()[38].sage()
-29472026335337227150423659490832640468979/
↪274214482066329363682430667508979749984665600000000

sage: ex = sqrt(atan(x)); a = fricas(ex).series(x=0); a
 1       5         9
 -       -         -
 2    1  2     31  2       6
x  - - x   + --- x   + O(x )
```

(continues on next page)

(continued from previous page)

```
     6      360

sage: a.coefficient(9/2).sage()
31/360

sage: x = fricas("x::TaylorSeries Fraction Integer")
sage: y = fricas("y::TaylorSeries Fraction Integer")
sage: 2*(1+2*x+sqrt(1-4*x)-2*x*y).recip()
              2        3      2 2       3        4         4         5
  1 + (x y + x ) + 2 x  + (x y  + 2 x y + 6 x ) + (4 x y + 18 x )
+
    3 3       4 2        5         6         5 2        6          7
  (x y  + 3 x y  + 13 x y + 57 x ) + (6 x y  + 40 x y + 186 x )
+
    4 4       5 3        6 2         7          8
  (x y  + 4 x y  + 21 x y  + 130 x y + 622 x )
+
      6 3        7 2         8           9
  (8 x y  + 66 x y  + 432 x y + 2120 x )
+
    5 5       6 4        7 3         8 2           9           10
  (x y  + 5 x y  + 30 x y  + 220 x y  + 1466 x y + 7338 x  ) + O(11)
```

```
>>> from sage.all import *
>>> x = var('x'); ex = sqrt(cos(x)); a = fricas(ex).series(x=Integer(0)); a
    1  2    1  4    19   6     559   8     29161    10      11
1 - - x  - -- x  - ---- x  - ------ x  - --------- x   + O(x  )
    4      96      5760      645120      116121600

>>> a.coefficients()[Integer(38)].sage()
-29472026335337227150423659490832640468979/
↪274214482066329363682430667508979749984665600000000

>>> ex = sqrt(atan(x)); a = fricas(ex).series(x=Integer(0)); a
 1       5         9
 -       -         -
 2    1  2     31  2       6
x  - - x  + --- x  + O(x )
     6      360

>>> a.coefficient(Integer(9)/Integer(2)).sage()
31/360

>>> x = fricas("x::TaylorSeries Fraction Integer")
>>> y = fricas("y::TaylorSeries Fraction Integer")
>>> Integer(2)*(Integer(1)+Integer(2)*x+sqrt(Integer(1)-Integer(4)*x)-Integer(2)*x*y).
↪recip()
              2        3      2 2       3        4         4         5
  1 + (x y + x ) + 2 x  + (x y  + 2 x y + 6 x ) + (4 x y + 18 x )
+
    3 3       4 2        5         6         5 2        6          7
  (x y  + 3 x y  + 13 x y + 57 x ) + (6 x y  + 40 x y + 186 x )
```

(continues on next page)

(continued from previous page)

```
+
    4 4       5 3        6 2         7          8
  (x y  + 4 x y  + 21 x y  + 130 x y + 622 x )
+
      6 3        7 2         8          9
  (8 x y  + 66 x y  + 432 x y + 2120 x )
+
    5 5       6 4        7 3         8 2          9          10
  (x y  + 5 x y  + 30 x y  + 220 x y  + 1466 x y + 7338 x   ) + O(11)
```

FriCAS does some limits right:

```
sage: x = var('x'); ex = x^2*exp(-x)*Ei(x) - x; fricas(ex).limit(x=oo)
1
```

```
>>> from sage.all import *
>>> x = var('x'); ex = x**Integer(2)*exp(-x)*Ei(x) - x; fricas(ex).limit(x=oo)
1
```

class sage.interfaces.fricas.**FriCAS**(*name='fricas'*, *command=None*, *script_subdirectory=None*, *logfile=None*, *server=None*, *server_tmpdir=None*)

Bases: `ExtraTabCompletion`, *Expect*

Interface to a FriCAS interpreter.

console()

Spawn a new FriCAS command-line session.

EXAMPLES:

```
sage: fricas.console()                                                    # not␣
↪tested
                   FriCAS (AXIOM fork) Computer Algebra System
                          Version: FriCAS 1.0.5
          Timestamp: Thursday February 19, 2009 at 06:57:33
-----------------------------------------------------------------------------
   Issue )copyright to view copyright notices.
   Issue )summary for a summary of useful system commands.
   Issue )quit to leave AXIOM and return to shell.
-----------------------------------------------------------------------------
```

```
>>> from sage.all import *
>>> fricas.console()                                                    # not tested
                   FriCAS (AXIOM fork) Computer Algebra System
                          Version: FriCAS 1.0.5
          Timestamp: Thursday February 19, 2009 at 06:57:33
-----------------------------------------------------------------------------
   Issue )copyright to view copyright notices.
   Issue )summary for a summary of useful system commands.
   Issue )quit to leave AXIOM and return to shell.
-----------------------------------------------------------------------------
```

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

Evaluate `code` using FriCAS.

Except `reformat`, all arguments are passed to *sage.interfaces.expect.Expect.eval()*.

INPUT:

- `reformat` – boolean; remove the output markers when True

This can also be used to pass system commands to FriCAS.

EXAMPLES:

```
sage: fricas.set("x", "1783"); fricas("x")
1783
sage: fricas.eval(")cl val x");
''
sage: fricas("x")
x
```

```
>>> from sage.all import *
>>> fricas.set("x", "1783"); fricas("x")
1783
>>> fricas.eval(")cl val x");
''
>>> fricas("x")
x
```

get (*var*)

Get the string representation of the value (more precisely, the OutputForm) of a variable or expression in FriCAS.

If FriCAS cannot evaluate var an error is raised.

EXAMPLES:

```
sage: fricas.set('xx', '2')
sage: fricas.get('xx')
'2'
sage: a = fricas('(1 + sqrt(2))^5')
sage: fricas.get(a.name())
'    +-+\n29 \\|2  + 41'
sage: fricas.get('(1 + sqrt(2))^5')
'    +-+\n29 \\|2  + 41'
sage: fricas.new('(1 + sqrt(2))^5')
    +-+
29 \|2  + 41
```

```
>>> from sage.all import *
>>> fricas.set('xx', '2')
>>> fricas.get('xx')
'2'
>>> a = fricas('(1 + sqrt(2))^5')
>>> fricas.get(a.name())
'    +-+\n29 \\|2  + 41'
>>> fricas.get('(1 + sqrt(2))^5')
```

(continues on next page)

(continued from previous page)

```
'    +-+\n29 \\|2  + 41'
>>> fricas.new('(1 + sqrt(2))^5')
    +-+
29 \|2  + 41
```

get_InputForm(*var*)

Return the `InputForm` as a string.

get_boolean(*var*)

Return the value of a FriCAS boolean as a boolean, without checking that it is a boolean.

get_integer(*var*)

Return the value of a FriCAS integer as an integer, without checking that it is an integer.

get_string(*var*)

Return the value of a FriCAS string as a string, without checking that it is a string.

get_unparsed_InputForm(*var*)

Return the unparsed `InputForm` as a string.

Todo

- catch errors, especially when InputForm is not available:
  - for example when integration returns `'failed'`
  - `UnivariatePolynomial`
- should we provide workarounds, too?

set(*var*, *value*)

Set a variable to a value in FriCAS.

INPUT:

- `var`, `value` – strings; the first representing a valid FriCAS variable identifier, the second a FriCAS expression

OUTPUT: none

EXAMPLES:

```
sage: fricas.set('xx', '2')
sage: fricas.get('xx')
'2'
```

```
>>> from sage.all import *
>>> fricas.set('xx', '2')
>>> fricas.get('xx')
'2'
```

class sage.interfaces.fricas.**FriCASElement**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: `ExpectElement`, `FriCASElement`

Instances of this class represent objects in FriCAS.

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

sage()

Convert `self` to a Sage object.

EXAMPLES:

Floats:

```
sage: fricas(2.1234).sage()
2.12340000000000
sage: _.parent()
Real Field with 53 bits of precision
sage: a = RealField(100)(pi)
sage: fricas(a).sage()
3.1415926535897932384626433833
sage: _.parent()
Real Field with 100 bits of precision
sage: fricas(a).sage() == a
True
sage: fricas(2.0).sage()
2.00000000000000
sage: _.parent()
Real Field with 53 bits of precision
```

```
>>> from sage.all import *
>>> fricas(RealNumber('2.1234')).sage()
2.12340000000000
>>> _.parent()
Real Field with 53 bits of precision
>>> a = RealField(Integer(100))(pi)
>>> fricas(a).sage()
3.1415926535897932384626433833
>>> _.parent()
Real Field with 100 bits of precision
>>> fricas(a).sage() == a
True
>>> fricas(RealNumber('2.0')).sage()
2.00000000000000
>>> _.parent()
Real Field with 53 bits of precision
```

Algebraic numbers:

```
sage: a = fricas('(1 + sqrt(2))^5'); a
    +-+
29 \|2  + 41
sage: b = a.sage(); b
82.0121933088198?
sage: b.radical_expression()
29*sqrt(2) + 41
```

```
>>> from sage.all import *
>>> a = fricas('(1 + sqrt(2))^5'); a
```

(continues on next page)

(continued from previous page)

```
    +-+
29 \|2  + 41
>>> b = a.sage(); b
82.0121933088198?
>>> b.radical_expression()
29*sqrt(2) + 41
```

Integers modulo n:

```
sage: fricas("((42^17)^1783)::IntegerMod(5^(5^5))").sage() == Integers(5^(5^
↪5))((42^17)^1783)
True
```

```
>>> from sage.all import *
>>> fricas("((42^17)^1783)::IntegerMod(5^(5^5))").sage() == Integers(Integer(5
↪)**(Integer(5)**Integer(5)))((Integer(42)**Integer(17))**Integer(1783))
True
```

Matrices over a prime field:

```
sage: fricas("matrix [[1::PF 3, 2],[2, 0]]").sage().parent()
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 3
```

```
>>> from sage.all import *
>>> fricas("matrix [[1::PF 3, 2],[2, 0]]").sage().parent()
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 3
```

We can also convert FriCAS's polynomials to Sage polynomials:

```
sage: a = fricas("x^2 + 1"); a.typeOf()
Polynomial(Integer)
sage: a.sage()
x^2 + 1
sage: _.parent()
Univariate Polynomial Ring in x over Integer Ring
sage: fricas('x^2 + y^2 + 1/2').sage()
y^2 + x^2 + 1/2
sage: _.parent()
Multivariate Polynomial Ring in y, x over Rational Field

sage: fricas("1$Polynomial Integer").sage()
1

sage: fricas("x^2/2").sage()
1/2*x^2

sage: x = polygen(QQ, 'x')
sage: fricas(x+3).sage()
x + 3
sage: fricas(x+3).domainOf()
Polynomial(Integer...)
```

(continues on next page)

(continued from previous page)

```
sage: fricas(matrix([[2,3],[4,x+5]])).diagonal().sage()
(2, x + 5)

sage: f = fricas("(y^2+3)::UP(y, INT)").sage(); f
y^2 + 3
sage: f.parent()
Univariate Polynomial Ring in y over Integer Ring

sage: fricas("(y^2+sqrt 3)::UP(y, AN)").sage()
y^2 + 1.732050807568878?
```

```
>>> from sage.all import *
>>> a = fricas("x^2 + 1"); a.typeOf()
Polynomial(Integer)
>>> a.sage()
x^2 + 1
>>> _.parent()
Univariate Polynomial Ring in x over Integer Ring
>>> fricas('x^2 + y^2 + 1/2').sage()
y^2 + x^2 + 1/2
>>> _.parent()
Multivariate Polynomial Ring in y, x over Rational Field

>>> fricas("1$Polynomial Integer").sage()
1

>>> fricas("x^2/2").sage()
1/2*x^2

>>> x = polygen(QQ, 'x')
>>> fricas(x+Integer(3)).sage()
x + 3
>>> fricas(x+Integer(3)).domainOf()
Polynomial(Integer...)

>>> fricas(matrix([[Integer(2),Integer(3)],[Integer(4),x+Integer(5)]])).
↪diagonal().sage()
(2, x + 5)

>>> f = fricas("(y^2+3)::UP(y, INT)").sage(); f
y^2 + 3
>>> f.parent()
Univariate Polynomial Ring in y over Integer Ring

>>> fricas("(y^2+sqrt 3)::UP(y, AN)").sage()
y^2 + 1.732050807568878?
```

Rational functions:

```
sage: fricas("x^2 + 1/z").sage()
x^2 + 1/z
```

```
>>> from sage.all import *
>>> fricas("x^2 + 1/z").sage()
x^2 + 1/z
```

Expressions:

```
sage: fricas(pi).sage()
pi

sage: fricas("sin(x+y)/exp(z)*log(1+%e)").sage()
e^(-z)*log(e + 1)*sin(x + y)

sage: fricas("factorial(n)").sage()
factorial(n)

sage: fricas("integrate(sin(x+y), x=0..1)").sage()
-cos(y + 1) + cos(y)

sage: fricas("integrate(x*sin(1/x), x=0..1)").sage()
'failed'

sage: fricas("integrate(sin((x^2+1)/x),x)").sage()
integral(sin((x^2 + 1)/x), x)
```

```
>>> from sage.all import *
>>> fricas(pi).sage()
pi

>>> fricas("sin(x+y)/exp(z)*log(1+%e)").sage()
e^(-z)*log(e + 1)*sin(x + y)

>>> fricas("factorial(n)").sage()
factorial(n)

>>> fricas("integrate(sin(x+y), x=0..1)").sage()
-cos(y + 1) + cos(y)

>>> fricas("integrate(x*sin(1/x), x=0..1)").sage()
'failed'

>>> fricas("integrate(sin((x^2+1)/x),x)").sage()
integral(sin((x^2 + 1)/x), x)
```

Todo

- Converting matrices and lists takes much too long.

Matrices:

```
sage: fricas("matrix [[x^n/2^m for n in 0..5] for m in 0..3]").sage()    #␣
↪long time
```

(continues on next page)

(continued from previous page)

```
[      1        x      x^2      x^3      x^4      x^5]
[     1/2   1/2*x 1/2*x^2 1/2*x^3 1/2*x^4 1/2*x^5]
[     1/4   1/4*x 1/4*x^2 1/4*x^3 1/4*x^4 1/4*x^5]
[     1/8   1/8*x 1/8*x^2 1/8*x^3 1/8*x^4 1/8*x^5]
```

```
>>> from sage.all import *
>>> fricas("matrix [[x^n/2^m for n in 0..5] for m in 0..3]").sage()    # long␣
↪time
[      1        x      x^2      x^3      x^4      x^5]
[     1/2   1/2*x 1/2*x^2 1/2*x^3 1/2*x^4 1/2*x^5]
[     1/4   1/4*x 1/4*x^2 1/4*x^3 1/4*x^4 1/4*x^5]
[     1/8   1/8*x 1/8*x^2 1/8*x^3 1/8*x^4 1/8*x^5]
```

Lists:

```
sage: fricas("[2^n/x^n for n in 0..5]").sage()                       # long␣
↪time
[1, 2/x, 4/x^2, 8/x^3, 16/x^4, 32/x^5]

sage: fricas("[matrix [[i for i in 1..n]] for n in 0..5]").sage()    # long␣
↪time
[[], [1], [1 2], [1 2 3], [1 2 3 4], [1 2 3 4 5]]
```

```
>>> from sage.all import *
>>> fricas("[2^n/x^n for n in 0..5]").sage()                       # long time
[1, 2/x, 4/x^2, 8/x^3, 16/x^4, 32/x^5]

>>> fricas("[matrix [[i for i in 1..n]] for n in 0..5]").sage()    # long time
[[], [1], [1 2], [1 2 3], [1 2 3 4], [1 2 3 4 5]]
```

Error handling:

```
sage: s = fricas.guessPade("[fibonacci i for i in 0..10]"); s
    n         x
[[[x ]- ----------]]
         2
        x  + x - 1
sage: s.sage()
Traceback (most recent call last):
...
NotImplementedError: the translation of the FriCAS Expression 'FEseries' to␣
↪sage is not yet implemented

sage: s = fricas("series(sqrt(1+x), x=0)"); s
      1     1  2    1  3     5   4     7   5    21   6    33   7    429   8
  1 + - x - - x  + -- x  - --- x  + --- x  - ---- x  + ---- x  - ----- x
      2     8      16      128      256      1024      2048      32768
+
   715   9    2431   10      11
  ----- x  - ------ x   + O(x  )
  65536       262144
```

(continues on next page)

(continued from previous page)

```
sage: s.sage()
Traceback (most recent call last):
...
NotImplementedError: the translation of the FriCAS object

      1     1  2    1  3     5   4    7   5     21   6    33   7     429   8
  1 + - x - - x  + -- x  - --- x  + --- x  - ---- x  + ---- x  - ----- x
      2     8      16      128      256      1024      2048      32768
+
   715   9    2431   10      11
  ----- x  - ------ x   + O(x   )
  65536      262144

to sage is not yet implemented:
An error occurred when FriCAS evaluated 'unparse(...::InputForm)':

   Cannot convert the value from type Any to InputForm .
```

```
>>> from sage.all import *
>>> s = fricas.guessPade("[fibonacci i for i in 0..10]"); s
    n        x
[[[x ]- ----------]]
         2
        x  + x - 1
>>> s.sage()
Traceback (most recent call last):
...
NotImplementedError: the translation of the FriCAS Expression 'FEseries' to
↪sage is not yet implemented

>>> s = fricas("series(sqrt(1+x), x=0)"); s
      1     1  2    1  3     5   4    7   5     21   6    33   7     429   8
  1 + - x - - x  + -- x  - --- x  + --- x  - ---- x  + ---- x  - ----- x
      2     8      16      128      256      1024      2048      32768
+
   715   9    2431   10      11
  ----- x  - ------ x   + O(x   )
  65536      262144

>>> s.sage()
Traceback (most recent call last):
...
NotImplementedError: the translation of the FriCAS object
<BLANKLINE>
      1     1  2    1  3     5   4    7   5     21   6    33   7     429   8
  1 + - x - - x  + -- x  - --- x  + --- x  - ---- x  + ---- x  - ----- x
      2     8      16      128      256      1024      2048      32768
+
   715   9    2431   10      11
  ----- x  - ------ x   + O(x   )
  65536      262144
<BLANKLINE>
```

(continues on next page)

(continued from previous page)

```
to sage is not yet implemented:
An error occurred when FriCAS evaluated 'unparse(...::InputForm)':
<BLANKLINE>
   Cannot convert the value from type Any to InputForm .
```

bool()

Coerce the expression into a boolean.

EXAMPLES:

```
sage: fricas("1=1").bool()
True
sage: fricas("1~=1").bool()
False
```

```
>>> from sage.all import *
>>> fricas("1=1").bool()
True
>>> fricas("1~=1").bool()
False
```

gen(*n*)

Return an error, since the n-th generator in FriCAS is not well defined.

class sage.interfaces.fricas.**FriCASExpectFunction**(*parent*, *name*)

Bases: *ExpectFunction*

Translate the pythonized function identifier back to a FriCAS operation name.

class sage.interfaces.fricas.**FriCASFunctionElement**(*object*, *name*)

Bases: *FunctionElement*

Make FriCAS operation names valid python function identifiers.

sage.interfaces.fricas.**fricas_console**()

Spawn a new FriCAS command-line session.

EXAMPLES:

```
sage: fricas_console()                                                  # not␣
↪tested
                 FriCAS (AXIOM fork) Computer Algebra System
                            Version: FriCAS 1.0.5
             Timestamp: Thursday February 19, 2009 at 06:57:33
-----------------------------------------------------------------------------
   Issue )copyright to view copyright notices.
   Issue )summary for a summary of useful system commands.
   Issue )quit to leave AXIOM and return to shell.
-----------------------------------------------------------------------------
```

```
>>> from sage.all import *
>>> fricas_console()                                                # not tested
                 FriCAS (AXIOM fork) Computer Algebra System
                            Version: FriCAS 1.0.5
```

(continues on next page)

(continued from previous page)

```
              Timestamp: Thursday February 19, 2009 at 06:57:33
-----------------------------------------------------------------------------
   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.fricas.**is_FriCASElement**(*x*)

Return `True` if `x` is of type *FriCASElement*.

EXAMPLES:

```
sage: from sage.interfaces.fricas import is_FriCASElement
sage: is_FriCASElement(2)
doctest:...: DeprecationWarning: the function is_FriCASElement is deprecated; use␣
↪isinstance(x, sage.interfaces.abc.FriCASElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
sage: is_FriCASElement(fricas(2))
True
```

```
>>> from sage.all import *
>>> from sage.interfaces.fricas import is_FriCASElement
>>> is_FriCASElement(Integer(2))
doctest:...: DeprecationWarning: the function is_FriCASElement is deprecated; use␣
↪isinstance(x, sage.interfaces.abc.FriCASElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
>>> is_FriCASElement(fricas(Integer(2)))
True
```

sage.interfaces.fricas.**reduce_load_fricas**()

Return the FriCAS interface object defined in *sage.interfaces.fricas*.

EXAMPLES:

```
sage: from sage.interfaces.fricas import reduce_load_fricas
sage: reduce_load_fricas()
FriCAS
```

```
>>> from sage.all import *
>>> from sage.interfaces.fricas import reduce_load_fricas
>>> reduce_load_fricas()
FriCAS
```

CHAPTER

NINE

INTERFACE TO FROBBY FOR FAST COMPUTATIONS ON MONOMIAL IDEALS.

The software package Frobby provides a number of computations on monomial ideals. The current main feature is the socle of a monomial ideal, which is largely equivalent to computing the maximal standard monomials, the Alexander dual or the irreducible decomposition.

Operations on monomial ideals are much faster than algorithms designed for ideals in general, which is what makes a specialized library for these operations on monomial ideals useful.

AUTHORS:

- Bjarke Hammersholt Roune (2008-04-25): Wrote the Frobby C++ program and the initial version of the Python interface.

Note

The official source for Frobby is <https://www.broune.com/frobby>, which also has documentation and papers describing the algorithms used.

class sage.interfaces.frobby.**Frobby**

Bases: object

alexander_dual(*monomial_ideal*)

This function computes the Alexander dual of the passed-in monomial ideal. This ideal is the one corresponding to the simplicial complex whose faces are the complements of the nonfaces of the simplicial complex corresponding to the input ideal.

INPUT:

- monomial_ideal – the monomial ideal to decompose

OUTPUT: the monomial corresponding to the Alexander dual

EXAMPLES:

This is a simple example of computing irreducible decomposition.

```
sage: # optional - frobby
sage: (a, b, c, d) = QQ['a,b,c,d'].gens()
sage: id = ideal(a * b, b * c, c * d, d * a)
sage: alexander_dual = frobby.alexander_dual(id)
sage: true_alexander_dual = ideal(b * d, a * c)
sage: alexander_dual == true_alexander_dual # use sets to ignore order
True
```

```
>>> from sage.all import *
>>> # optional - frobby
>>> (a, b, c, d) = QQ['a,b,c,d'].gens()
>>> id = ideal(a * b, b * c, c * d, d * a)
>>> alexander_dual = frobby.alexander_dual(id)
>>> true_alexander_dual = ideal(b * d, a * c)
>>> alexander_dual == true_alexander_dual # use sets to ignore order
True
```

We see how it is much faster to compute this with frobby than the built-in procedure for simplicial complexes:

```
sage: # optional - frobby
sage: t=simplicial_complexes.PoincareHomologyThreeSphere()
sage: R=PolynomialRing(QQ,16,'x')
sage: I=R.ideal([prod([R.gen(i-1) for i in a]) for a in t.facets()])
sage: len(frobby.alexander_dual(I).gens())
643
```

```
>>> from sage.all import *
>>> # optional - frobby
>>> t=simplicial_complexes.PoincareHomologyThreeSphere()
>>> R=PolynomialRing(QQ,Integer(16),'x')
>>> I=R.ideal([prod([R.gen(i-Integer(1)) for i in a]) for a in t.facets()])
>>> len(frobby.alexander_dual(I).gens())
643
```

associated_primes(*monomial_ideal*)

This function computes the associated primes of the passed-in monomial ideal.

INPUT:

- `monomial_ideal` – the monomial ideal to decompose

OUTPUT:

A list of the associated primes of the monomial ideal. These ideals are constructed in the same ring as monomial_ideal is.

EXAMPLES:

```
sage: R.<d,b,c>=QQ[] # optional - frobby
sage: I=[d*b*c,b^2*c,b^10,d^10]*R # optional - frobby
sage: frobby.associated_primes(I)   # optional - frobby
[Ideal (d, b) of Multivariate Polynomial Ring in d, b, c over Rational Field,
Ideal (d, b, c) of Multivariate Polynomial Ring in d, b, c over Rational␣
↪Field]
```

```
>>> from sage.all import *
>>> R = QQ['d, b, c']; (d, b, c,) = R._first_ngens(3)# optional - frobby
>>> I=[d*b*c,b**Integer(2)*c,b**Integer(10),d**Integer(10)]*R # optional -␣
↪frobby
>>> frobby.associated_primes(I)   # optional - frobby
[Ideal (d, b) of Multivariate Polynomial Ring in d, b, c over Rational Field,
Ideal (d, b, c) of Multivariate Polynomial Ring in d, b, c over Rational␣
↪Field]
```

dimension(*monomial_ideal*)

This function computes the dimension of the passed-in monomial ideal.

INPUT:

- `monomial_ideal` – the monomial ideal to decompose

OUTPUT: the dimension of the zero set of the ideal

EXAMPLES:

```
sage: R.<d,b,c>=QQ[] # optional - frobby
sage: I=[d*b*c,b^2*c,b^10,d^10]*R # optional - frobby
sage: frobby.dimension(I)    # optional - frobby
1
```

```
>>> from sage.all import *
>>> R = QQ['d, b, c']; (d, b, c,) = R._first_ngens(3)# optional - frobby
>>> I=[d*b*c,b**Integer(2)*c,b**Integer(10),d**Integer(10)]*R # optional -
↪frobby
>>> frobby.dimension(I)    # optional - frobby
1
```

hilbert(*monomial_ideal*)

Compute the multigraded Hilbert-Poincaré series of the input ideal. Use the -univariate option to get the univariate series.

The Hilbert-Poincaré series of a monomial ideal is the sum of all monomials not in the ideal. This sum can be written as a (finite) rational function with $(x_1 - 1)(x_2 - 1)...(x_n - 1)$ in the denominator, assuming the variables of the ring are $x_1, x2, ..., x_n$. This action computes the polynomial in the numerator of this fraction.

INPUT:

- `monomial_ideal` – a monomial ideal

OUTPUT:

A polynomial in the same ring as the ideal.

EXAMPLES:

```
sage: R.<d,b,c>=QQ[] # optional - frobby
sage: I=[d*b*c,b^2*c,b^10,d^10]*R # optional - frobby
sage: frobby.hilbert(I) # optional - frobby
d^10*b^10*c + d^10*b^10 + d^10*b*c + b^10*c + d^10 + b^10 + d*b^2*c + d*b*c +
↪b^2*c + 1
```

```
>>> from sage.all import *
>>> R = QQ['d, b, c']; (d, b, c,) = R._first_ngens(3)# optional - frobby
>>> I=[d*b*c,b**Integer(2)*c,b**Integer(10),d**Integer(10)]*R # optional -
↪frobby
>>> frobby.hilbert(I) # optional - frobby
d^10*b^10*c + d^10*b^10 + d^10*b*c + b^10*c + d^10 + b^10 + d*b^2*c + d*b*c +
↪b^2*c + 1
```

irreducible_decomposition(*monomial_ideal*)

This function computes the irreducible decomposition of the passed-in monomial ideal. I.e. it computes the unique minimal list of irreducible monomial ideals whose intersection equals monomial_ideal.

INPUT:

- `monomial_ideal` – the monomial ideal to decompose

OUTPUT:

A list of the unique irredundant irreducible components of monomial_ideal. These ideals are constructed in the same ring as monomial_ideal is.

EXAMPLES:

This is a simple example of computing irreducible decomposition.

```
sage: # optional - frobby
sage: (x, y, z) = QQ['x,y,z'].gens()
sage: id = ideal(x ** 2, y ** 2, x * z, y * z)
sage: decom = frobby.irreducible_decomposition(id)
sage: true_decom = [ideal(x, y), ideal(x ** 2, y ** 2, z)]
sage: set(decom) == set(true_decom) # use sets to ignore order
True
```

```
>>> from sage.all import *
>>> # optional - frobby
>>> (x, y, z) = QQ['x,y,z'].gens()
>>> id = ideal(x ** Integer(2), y ** Integer(2), x * z, y * z)
>>> decom = frobby.irreducible_decomposition(id)
>>> true_decom = [ideal(x, y), ideal(x ** Integer(2), y ** Integer(2), z)]
>>> set(decom) == set(true_decom) # use sets to ignore order
True
```

We now try the special case of the zero ideal in different rings.

We should also try PolynomialRing(QQ, names=[]), but it has a bug which makes that impossible (see Issue #3028).

```
sage: # optional - frobby
sage: rings = [ZZ['x'], CC['x,y']]
sage: allOK = True
sage: for ring in rings:
....:     id0 = ring.ideal(0)
....:     decom0 = frobby.irreducible_decomposition(id0)
....:     allOK = allOK and decom0 == [id0]
sage: allOK
True
```

```
>>> from sage.all import *
>>> # optional - frobby
>>> rings = [ZZ['x'], CC['x,y']]
>>> allOK = True
>>> for ring in rings:
...     id0 = ring.ideal(Integer(0))
...     decom0 = frobby.irreducible_decomposition(id0)
...     allOK = allOK and decom0 == [id0]
>>> allOK
True
```

Finally, we try the ideal that is all of the ring in different rings.

```
sage: # optional - frobby
sage: rings = [ZZ['x'], CC['x,y']]
sage: allOK = True
sage: for ring in rings:
....:     id1 = ring.ideal(1)
....:     decom1 = frobby.irreducible_decomposition(id1)
....:     allOK = allOK and decom1 == [id1]
sage: allOK
True
```

```
>>> from sage.all import *
>>> # optional - frobby
>>> rings = [ZZ['x'], CC['x,y']]
>>> allOK = True
>>> for ring in rings:
...     id1 = ring.ideal(Integer(1))
...     decom1 = frobby.irreducible_decomposition(id1)
...     allOK = allOK and decom1 == [id1]
>>> allOK
True
```

CHAPTER

TEN

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.

10.1 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
```

```
>>> from sage.all import *
>>> n = gap(Integer(20062006)); n
20062006
>>> n.parent()
Gap
>>> fac = n.Factors(); fac
[ 2, 17, 59, 73, 137 ]
>>> fac.parent()
Gap
>>> fac[Integer(1)]
2
```

10.2 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)
[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
```

```
>>> from sage.all import *
>>> singular(Integer(389))
389
>>> R1 = singular.ring(Integer(0), '(x,y)', 'dp')
>>> 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')
>>> F = f.factorize()
>>> print(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
```

Next we convert the factor $-x^5 + y^2$ to a Sage multivariate polynomial. Note that it is important to let x and y be the generators of a polynomial ring, so the eval command works.

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: s = F[1][3].sage_polystring(); s
'-x**5+y**2'
sage: g = eval(s); g
-x^5 + y^2
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R._first_ngens(2)
>>> s = F[Integer(1)][Integer(3)].sage_polystring(); s
'-x**5+y**2'
>>> g = eval(s); g
-x^5 + y^2
```

Next we create a polynomial ring in GAP and obtain its indeterminates:

```
sage: R = gap.PolynomialRing('Rationals', 2); R
```

(continues on next page)

(continued from previous page)

```
PolynomialRing( Rationals, ["x_1", "x_2"] )
sage: I = R.IndeterminatesOfPolynomialRing(); I
[ x_1, x_2 ]
```

```
>>> from sage.all import *
>>> R = gap.PolynomialRing('Rationals', Integer(2)); R
PolynomialRing( Rationals, ["x_1", "x_2"] )
>>> I = R.IndeterminatesOfPolynomialRing(); I
[ x_1, x_2 ]
```

In order to eval g in GAP, we need to tell GAP to view the variables `x0` and `x1` as the two generators of R. This is the one tricky part. In the GAP interpreter the object `I` has its own name (which isn't `I`). We can access its name using `I.name()`.

```
sage: _ = gap.eval("x := %s[1];; y := %s[2];;"%(I.name(), I.name()))
```

```
>>> from sage.all import *
>>> _ = gap.eval("x := %s[1];; y := %s[2];;"%(I.name(), I.name()))
```

Now x_0 and x_1 are defined, so we can construct the GAP polynomial f corresponding to g:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: f = gap(str(g)); f
-x_1^5+x_2^2
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R._first_ngens(2)
>>> f = gap(str(g)); f
-x_1^5+x_2^2
```

We can call GAP functions on f. For example, we evaluate the GAP `Value` function, which evaluates f at the point $(1, 2)$.

```
sage: f.Value(I, [1,2])
3
sage: g(1,2)          # agrees
3
```

```
>>> from sage.all import *
>>> f.Value(I, [Integer(1),Integer(2)])
3
>>> g(Integer(1),Integer(2))          # agrees
3
```

10.3 Saving and loading objects

Saving and loading GAP objects (using the dumps method, etc.) is *not* supported, since the output string representation of Gap objects is sometimes not valid input to GAP. Creating classes that wrap GAP objects *is* supported, via simply defining the a _gap_init_ member function that returns a string that when evaluated in GAP constructs the object. See `groups/perm_gps/permgroup.py` for a nontrivial example of this.

10.4 Long Input

The GAP interface reads in even very long input (using files) in a robust manner, as long as you are creating a new object.

Note

Using `gap.eval` for long input is much less robust, and is not recommended.

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

```
>>> from sage.all import *
>>> t = '"%s"'%Integer(10)**Integer(10000)   # ten thousand character string.
>>> a = gap(t)
```

10.5 Changing which GAP is used, and how

Set the environment variable `SAGE_GAP_COMMAND` to specify how GAP executable is called. E.g.

```
$ SAGE_GAP_COMMAND = "/usr/local/bin/gap -s 4G" ./sage
```

will use GAP installed in $/usr/local/bin$, with 4Gb RAM.

Set the environment variable `SAGE_GAP_MEMORY` to specify the amount of RAM allocated to `libgap` and to the GAP executable. If `SAGE_GAP_COMMAND` is set, as well, then `SAGE_GAP_MEMORY` is only used for $libgap$.

```
sage: gap.eval('GAPInfo.CommandLineOptions.s') # not tested
'"42m"'
```

```
>>> from sage.all import *
>>> gap.eval('GAPInfo.CommandLineOptions.s') # not tested
'"42m"'
```

After the GAP interface initialisation, setting `SAGE_GAP_MEMORY` has no effect:

```
sage: os.environ['SAGE_GAP_MEMORY'] = '24M'
sage: gap.eval('GAPInfo.CommandLineOptions.s') # not tested
'"4g"'
```

```
>>> from sage.all import *
>>> os.environ['SAGE_GAP_MEMORY'] = '24M'
>>> gap.eval('GAPInfo.CommandLineOptions.s') # not tested
'"4g"'
```

AUTHORS:

- David Joyner and William Stein: initial version(s)
- William Stein (2006-02-01): modified gap_console command so it uses exactly the same startup command as Gap.__init__.
- William Stein (2006-03-02): added tab completions: gap.[tab], x = gap(…), x.[tab], and docs, e.g., gap.function? and x.function?

class sage.interfaces.gap.**Gap**(*max_workspace_size=None, maxread=None, script_subdirectory=None, use_workspace_cache=True, server=None, server_tmpdir=None, logfile=None, seed=None, env={}*)

Bases: *Gap_generic*

Interface to the GAP interpreter.

AUTHORS:

- William Stein and David Joyner

console()

Spawn a new GAP command-line session.

EXAMPLES:

```
sage: gap.console()  # not tested
*********   GAP, Version 4.5.7 of 14-Dec-2012 (free software, GPL)
*  GAP  *   https://www.gap-system.org
*********   Architecture: x86_64-unknown-linux-gnu-gcc-default64
Libs used:  gmp, readline
Loading the library and packages ...
Packages:   GAPDoc 1.5.1
Try '?help' for help. See also  '?copyright' and  '?authors'
gap>
```

```
>>> from sage.all import *
>>> gap.console()  # not tested
*********   GAP, Version 4.5.7 of 14-Dec-2012 (free software, GPL)
*  GAP  *   https://www.gap-system.org
*********   Architecture: x86_64-unknown-linux-gnu-gcc-default64
Libs used:  gmp, readline
Loading the library and packages ...
Packages:   GAPDoc 1.5.1
Try '?help' for help. See also  '?copyright' and  '?authors'
gap>
```

cputime(*t=None*)

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

EXAMPLES:

```
sage: t = gap.cputime()
sage: t  #random
0.13600000000000001
sage: gap.Order(gap.SymmetricGroup(5))
120
sage: gap.cputime(t)  #random
0.059999999999999998
```

```
>>> from sage.all import *
>>> t = gap.cputime()
>>> t  #random
0.13600000000000001
```

(continues on next page)

(continued from previous page)

```
>>> gap.Order(gap.SymmetricGroup(Integer(5)))
120
>>> gap.cputime(t)  #random
0.05999999999999998
```

get (*var*, *use_file=False*)

Get the string representation of the variable var.

EXAMPLES:

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

```
>>> from sage.all import *
>>> gap.set('x', '2')
>>> gap.get('x')
'2'
```

help (*s*, *pager=True*)

Print help on a given topic.

EXAMPLES:

Note: In order to ensure consistent unicode handling from GAP we start a GAP instance with a forced UTF-8 locale:

```
sage: gap = Gap(env={'LC_CTYPE': 'en_US.UTF-8'})
sage: print(gap.help('SymmetricGroup', pager=False))

  50.1-... SymmetricGroup

  ‣ SymmetricGroup( [filt, ]deg ) ───────────────────────────────────── function
...
```

```
>>> from sage.all import *
>>> gap = Gap(env={'LC_CTYPE': 'en_US.UTF-8'})
>>> print(gap.help('SymmetricGroup', pager=False))
<BLANKLINE>
  50.1-... SymmetricGroup
<BLANKLINE>
  ‣ SymmetricGroup( [filt, ]deg ) ───────────────────────────────────── function
...
<BLANKLINE>
```

save_workspace ()

Save the GAP workspace.

set (*var*, *value*)

Set the variable var to the given value.

EXAMPLES:

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

```
>>> from sage.all import *
>>> gap.set('x', '2')
>>> gap.get('x')
'2'
```

set_seed(*seed=None*)

Set the seed for gap interpreter.

The seed should be an integer.

EXAMPLES:

```
sage: g = Gap()
sage: g.set_seed(0)
0
sage: [g.Random(1,10) for i in range(5)]
[2, 3, 3, 4, 2]
```

```
>>> from sage.all import *
>>> g = Gap()
>>> g.set_seed(Integer(0))
0
>>> [g.Random(Integer(1),Integer(10)) for i in range(Integer(5))]
[2, 3, 3, 4, 2]
```

class sage.interfaces.gap.**GapElement**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: *GapElement_generic*, *GapElement*

str(*use_file=False*)

EXAMPLES:

```
sage: print(gap(2))
2
```

```
>>> from sage.all import *
>>> print(gap(Integer(2)))
2
```

class sage.interfaces.gap.**GapElement_generic**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: ModuleElement, ExtraTabCompletion, *ExpectElement*

Generic interface to the GAP3/GAP4 interpreters.

AUTHORS:

- William Stein and David Joyner (interface for GAP4)
- Franco Saliola (Feb 2010): refactored to separate out the generic code

is_string()

Tell whether this element is a string.

EXAMPLES:

```
sage: gap('"abc"').is_string()
True
sage: gap('[1,2,3]').is_string()
False
```

```
>>> from sage.all import *
>>> gap('"abc"').is_string()
True
>>> gap('[1,2,3]').is_string()
False
```

class sage.interfaces.gap.**GapFunction**(*parent*, *name*)

Bases: *ExpectFunction*

class sage.interfaces.gap.**GapFunctionElement**(*obj*, *name*)

Bases: *FunctionElement*

class sage.interfaces.gap.**Gap_generic**(*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: ExtraTabCompletion, *Expect*

Generic interface to the GAP3/GAP4 interpreters.

AUTHORS:

- William Stein and David Joyner (interface for GAP4)
- Franco Saliola (Feb 2010): refactored to separate out the generic code

eval(*x*, *newlines=False*, *strip=True*, *split_lines=True*, ***kwds*)

Send the code in the string s to the GAP interpreter and return the output as a string.

INPUT:

- s – string containing GAP code
- newlines – boolean (default: True); if False, remove all backslash-newlines inserted by the GAP output formatter
- strip – ignored
- split_lines – boolean (default: True); if True then each line is evaluated separately. If False, then the whole block of code is evaluated all at once.

EXAMPLES:

```
sage: gap.eval('2+2')
'4'
sage: gap.eval('Print(4); #test\n Print(6);')
'46'
sage: gap.eval('Print("#"); Print(6);')
'#6'
sage: gap.eval('4; \n 6;')
'4\n6'
```

(continues on next page)

(continued from previous page)

```
sage: gap.eval('if 3>2 then\nPrint("hi");\nfi;')
'hi'
sage: gap.eval('## this is a test\nPrint("OK")')
'OK'
sage: gap.eval('Print("This is a test. Oh no, a #");# but this is a comment\
↪nPrint("OK")')
'This is a test. Oh no, a #OK'
sage: gap.eval('if 4>3 then')
''
sage: gap.eval('Print("Hi how are you?")')
'Hi how are you?'
sage: gap.eval('fi')
''
```

```
>>> from sage.all import *
>>> gap.eval('2+2')
'4'
>>> gap.eval('Print(4); #test\n Print(6);')
'46'
>>> gap.eval('Print("#"); Print(6);')
'#6'
>>> gap.eval('4; \n 6;')
'4\n6'
>>> gap.eval('if 3>2 then\nPrint("hi");\nfi;')
'hi'
>>> gap.eval('## this is a test\nPrint("OK")')
'OK'
>>> gap.eval('Print("This is a test. Oh no, a #");# but this is a comment\
↪nPrint("OK")')
'This is a test. Oh no, a #OK'
>>> gap.eval('if 4>3 then')
''
>>> gap.eval('Print("Hi how are you?")')
'Hi how are you?'
>>> gap.eval('fi')
''
```

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

Call the GAP function with `args` and `kwds`.

EXAMPLES:

```
sage: gap.function_call('SymmetricGroup', [5])
SymmetricGroup( [ 1 .. 5 ] )
```

```
>>> from sage.all import *
>>> gap.function_call('SymmetricGroup', [Integer(5)])
SymmetricGroup( [ 1 .. 5 ] )
```

If the GAP function does not return a value, but prints something to the screen, then a string of the printed output is returned.

```
sage: s = gap.function_call('Display', [gap.SymmetricGroup(5).
↪CharacterTable()])
sage: type(s)
<class 'sage.interfaces.interface.AsciiArtString'>
sage: s.startswith('CT')
True
```

```
>>> from sage.all import *
>>> s = gap.function_call('Display', [gap.SymmetricGroup(Integer(5)).
↪CharacterTable()])
>>> type(s)
<class 'sage.interfaces.interface.AsciiArtString'>
>>> s.startswith('CT')
True
```

get_record_element (*record*, *name*)

Return the element of a GAP record identified by `name`.

INPUT:

- `record` – a GAP record
- `name` – string

OUTPUT: *GapElement*

EXAMPLES:

```
sage: rec = gap('rec( a := 1, b := "2" )')
sage: gap.get_record_element(rec, 'a')
1
sage: gap.get_record_element(rec, 'b')
2
```

```
>>> from sage.all import *
>>> rec = gap('rec( a := 1, b := "2" )')
>>> gap.get_record_element(rec, 'a')
1
>>> gap.get_record_element(rec, 'b')
2
```

interrupt (*tries=None*, *timeout=1*, *quit_on_fail=True*)

Interrupt the GAP process.

Gap installs a SIGINT handler, we call it directly instead of trying to sent Ctrl-C. Unlike *interrupt()*, we only try once since we are knowing what we are doing.

Sometimes GAP dies while interrupting.

EXAMPLES:

```
sage: gap._eval_line('while(1=1) do i:=1;; od;', wait_for_prompt=False)
''
sage: rc = gap.interrupt(timeout=1)
sage: [ gap(i) for i in range(10) ]   # check that it is still working
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> from sage.all import *
>>> gap._eval_line('while(1=1) do i:=1;; od;', wait_for_prompt=False)
''
>>> rc = gap.interrupt(timeout=Integer(1))
>>> [ gap(i) for i in range(Integer(10)) ]   # check that it is still working
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

load_package(*pkg*, *verbose=False*)

Load the Gap package with the given name.

If loading fails, raise a `RuntimeError` exception.

unbind(*var*)

Clear the variable named var.

EXAMPLES:

```
sage: gap.set('x', '2')
sage: gap.get('x')
'2'
sage: gap.unbind('x')
sage: gap.get('x')
Traceback (most recent call last):
...
RuntimeError: Gap produced error output
Error, Variable: 'x' must have a value
...
```

```
>>> from sage.all import *
>>> gap.set('x', '2')
>>> gap.get('x')
'2'
>>> gap.unbind('x')
>>> gap.get('x')
Traceback (most recent call last):
...
RuntimeError: Gap produced error output
Error, Variable: 'x' must have a value
...
```

version()

Return the version of GAP being used.

EXAMPLES:

```
sage: print(gap.version())
4...
```

```
>>> from sage.all import *
>>> print(gap.version())
4...
```

sage.interfaces.gap.**gap_command**(*use_workspace_cache=True*, *local=True*) sage.interfaces.gap.**gap_console**()

Spawn a new GAP command-line session.

Note that in gap-4.5.7 you cannot use a workspace cache that had no commandline to restore a gap session with commandline.

EXAMPLES:

```
sage: gap_console()  # not tested
*********   GAP, Version 4.5.7 of 14-Dec-2012 (free software, GPL)
*  GAP   *   https://www.gap-system.org
*********   Architecture: x86_64-unknown-linux-gnu-gcc-default64
Libs used:  gmp, readline
Loading the library and packages ...
Packages:   GAPDoc 1.5.1
Try '?help' for help. See also  '?copyright' and  '?authors'
gap>
```

```
>>> from sage.all import *
>>> gap_console()  # not tested
*********   GAP, Version 4.5.7 of 14-Dec-2012 (free software, GPL)
*  GAP   *   https://www.gap-system.org
*********   Architecture: x86_64-unknown-linux-gnu-gcc-default64
Libs used:  gmp, readline
Loading the library and packages ...
Packages:   GAPDoc 1.5.1
Try '?help' for help. See also  '?copyright' and  '?authors'
gap>
```

sage.interfaces.gap.**gap_reset_workspace**(*max_workspace_size=None*, *verbose=False*)

Call this to completely reset the GAP workspace, which is used by default when Sage first starts GAP.

The first time you start GAP from Sage, it saves the startup state of GAP in a file `$HOME/.sage/gap/workspace-gap-HASH`, where `HASH` is a hash of the directory where Sage is installed. This is useful because the subsequent startup of GAP is at least ten times as fast. But if you update GAP or any of its packages, those changes won't take effect until the workspace is reset.

sage.interfaces.gap.**gfq_gap_to_sage**(*x*, *F*)

INPUT:

- `x` – GAP finite field element
- `F` – Sage finite field

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)
```

(continues on next page)

(continued from previous page)

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

```
>>> from sage.all import *
>>> x = gap('Z(13)')
>>> F = GF(Integer(13), 'a')
>>> F(x)
2
>>> F(gap('0*Z(13)'))
0
>>> F = GF(Integer(13)**Integer(2), 'a')
>>> x = gap('Z(13)')
>>> F(x)
2
>>> x = gap('Z(13^2)^3')
>>> F(x)
12*a + 11
>>> F.multiplicative_generator()**Integer(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()
```

(continues on next page)

(continued from previous page)

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

```
>>> from sage.all import *
>>> a = gap(Mod(Integer(3), Integer(18))); a
ZmodnZObj( 3, 18 )
>>> b = sage.interfaces.gap.intmod_gap_to_sage(a); b
3
>>> b.parent()
Ring of integers modulo 18

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

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

>>> a = gap(Mod(Integer(3), Integer(65537))); a
ZmodpZObj( 3, 65537 )
>>> b = sage.interfaces.gap.intmod_gap_to_sage(a); b
3
>>> b.parent()
Ring of integers modulo 65537
```

`sage.interfaces.gap.`**`is_GapElement`**`(x)`

Return `True` if `x` is a *`GapElement`*.

This function is deprecated; use `isinstance()` (of *`sage.interfaces.abc.GapElement`*) instead.

EXAMPLES:

```
sage: from sage.interfaces.gap import is_GapElement
sage: is_GapElement(gap(2))
doctest:...: DeprecationWarning: the function is_GapElement is deprecated; use
↪isinstance(x, sage.interfaces.abc.GapElement) instead
See https://github.com/sagemath/sage/issues/34823 for details.
True
sage: is_GapElement(2)
False
```

```
>>> from sage.all import *
>>> from sage.interfaces.gap import is_GapElement
>>> is_GapElement(gap(Integer(2)))
doctest:...: DeprecationWarning: the function is_GapElement is deprecated; use␣
↪isinstance(x, sage.interfaces.abc.GapElement) instead
See https://github.com/sagemath/sage/issues/34823 for details.
True
>>> is_GapElement(Integer(2))
False
```

sage.interfaces.gap.**reduce_load_GAP**()

Return the GAP interface object defined in `sage.interfaces.gap`.

EXAMPLES:

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

```
>>> from sage.all import *
>>> from sage.interfaces.gap import reduce_load_GAP
>>> reduce_load_GAP()
Gap
```

CHAPTER

ELEVEN

INTERFACE TO GAP3

This module implements an interface to GAP3.

AUTHORS:

- Franco Saliola (February 2010)
- Christian Stump (March 2016)

> **Warning**
>
> The experimental package for GAP3 is Jean Michel's pre-packaged GAP3, which is a minimal GAP3 distribution containing packages that have no equivalent in GAP4, see Issue #20107 and also
>
> https://webusers.imj-prg.fr/~jean.michel/gap3/

11.1 Obtaining GAP3

Instead of installing the experimental GAP3 package, one can as well install by hand either of the following two versions of GAP3:

- Frank Luebeck maintains a GAP3 Linux executable, optimized for i686 and statically linked for jobs of 2 GByte or more:

  http://www.math.rwth-aachen.de/~Frank.Luebeck/gap/GAP3

- or you can download GAP3 from the GAP website below. Since GAP3 is no longer supported, it may not be easy to install this version.

  https://www.gap-system.org/Gap3/Download3/download.html

11.2 Changing which GAP3 is used

> **Warning**
>
> There is a bug in the pexpect module (see Issue #8471) that prevents the following from working correctly. For now, just make sure that `gap3` is in your `PATH`.

Sage assumes that GAP3 can be launched with the command `gap3`; that is, Sage assumes that the command `gap3` is in your `PATH`. If this is not the case, then you can start GAP3 using the following command:

```
sage: gap3 = Gap3(command='/usr/local/bin/gap3')            # not tested
```

```
>>> from sage.all import *
>>> gap3 = Gap3(command='/usr/local/bin/gap3')            # not tested
```

11.3 Functionality and Examples

The interface to GAP3 offers the following functionality.

1. `gap3(expr)` – evaluation of arbitrary GAP3 expressions, with the result returned as a Sage object wrapping the corresponding GAP3 element:

```
sage: # optional - gap3
sage: a = gap3('3+2')
sage: a
5
sage: type(a)
<class 'sage.interfaces.gap3.GAP3Element'>
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> a = gap3('3+2')
>>> a
5
>>> type(a)
<class 'sage.interfaces.gap3.GAP3Element'>
```

```
sage: # optional - gap3
sage: S5 = gap3('SymmetricGroup(5)')
sage: S5
Group( (1,5), (2,5), (3,5), (4,5) )
sage: type(S5)
<class 'sage.interfaces.gap3.GAP3Record'>
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> S5 = gap3('SymmetricGroup(5)')
>>> S5
Group( (1,5), (2,5), (3,5), (4,5) )
>>> type(S5)
<class 'sage.interfaces.gap3.GAP3Record'>
```

This provides a Pythonic interface to GAP3. If `gap_function` is the name of a GAP3 function, then the syntax `gap_element.gap_function()` returns the `gap_element` obtained by evaluating the command `gap_function(gap_element)` in GAP3:

```
sage: # optional - gap3
sage: S5.Size()
120
sage: S5.CharTable()
CharTable( Group( (1,5), (2,5), (3,5), (4,5) ) )
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> S5.Size()
120
>>> S5.CharTable()
CharTable( Group( (1,5), (2,5), (3,5), (4,5) ) )
```

Alternatively, you can instead use the syntax `gap3.gap_function(gap_element)`:

```
sage: gap3.DerivedSeries(S5)                         #optional - gap3
[ Group( (1,5), (2,5), (3,5), (4,5) ),
  Subgroup( Group( (1,5), (2,5), (3,5), (4,5) ),
            [ (1,2,5), (1,3,5), (1,4,5) ] ) ]
```

```
>>> from sage.all import *
>>> gap3.DerivedSeries(S5)                         #optional - gap3
[ Group( (1,5), (2,5), (3,5), (4,5) ),
  Subgroup( Group( (1,5), (2,5), (3,5), (4,5) ),
            [ (1,2,5), (1,3,5), (1,4,5) ] ) ]
```

If `gap_element` corresponds to a GAP3 record, then `gap_element.recfield` provides a means to access the record element corresponding to the field `recfield`:

```
sage: # optional - gap3
sage: S5.IsRec()
true
sage: S5.recfields()
['isDomain', 'isGroup', 'identity', 'generators', 'operations',
'isPermGroup', 'isFinite', '1', '2', '3', '4', 'degree']
sage: S5.identity
()
sage: S5.degree
5
sage: S5.1
(1,5)
sage: S5.2
(2,5)
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> S5.IsRec()
true
>>> S5.recfields()
['isDomain', 'isGroup', 'identity', 'generators', 'operations',
'isPermGroup', 'isFinite', '1', '2', '3', '4', 'degree']
>>> S5.identity
()
>>> S5.degree
5
>>> S5.gen(1)
(1,5)
>>> S5.gen(2)
(2,5)
```

2. By typing `%gap3` or `gap3.interact()` at the command-line, you can interact directly with the underlying GAP3 session.

```
sage: gap3.interact()                               # not tested

  --> Switching to Gap3 <--

gap3:
```

```
>>> from sage.all import *
>>> gap3.interact()                               # not tested

  --> Switching to Gap3 <--

gap3:
```

3. You can start a new GAP3 session as follows:

```
sage: gap3.console()                                # not tested

             ########            Lehrstuhl D fuer Mathematik
           ###    ####           RWTH Aachen
          ##         ##
         ##          #             #######            #########
        ##                        #      ##          ## #     ##
        ##           #           #       ##             #      ##
        ####        ##           ##       #             #      ##
         #####     ###           ##      ##             ##    ##
           ######### #            #########             #######
                     #                                  #
                    ##           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>
```

```
>>> from sage.all import *
>>> gap3.console()                                # not tested

             ########            Lehrstuhl D fuer Mathematik
           ###    ####           RWTH Aachen
          ##         ##
         ##          #             #######            #########
        ##                        #      ##          ## #     ##
        ##           #           #       ##             #      ##
```

(continues on next page)

(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>
```

4. The interface also has access to the GAP3 help system:

```
sage: gap3.help('help', pager=False)             # not tested
Help _______________________________________________________...

This  section describes  together with  the following sections the   GAP
help system.  The help system lets you read the manual interactively...
```

```
>>> from sage.all import *
>>> gap3.help('help', pager=False)             # not tested
Help _______________________________________________________...

This  section describes  together with  the following sections the   GAP
help system.  The help system lets you read the manual interactively...
```

11.4 Common Pitfalls

1. If you want to pass a string to GAP3, then you need to wrap it in single quotes as follows:

```
sage: gap3('"This is a GAP3 string"')              #optional - gap3
"This is a GAP3 string"
```

```
>>> from sage.all import *
>>> gap3('"This is a GAP3 string"')              #optional - gap3
"This is a GAP3 string"
```

This is particularly important when a GAP3 package is loaded via the `RequirePackage` method (note that one can instead use the `load_package` method):

```
sage: gap3.RequirePackage('"chevie"')              #optional - gap3
```

```
>>> from sage.all import *
>>> gap3.RequirePackage('"chevie"')              #optional - gap3
```

11.5 Examples

Load a GAP3 package:

```
sage: # optional - gap3
sage: gap3.load_package("chevie")
sage: gap3.version() # random  # not tested
'lib: v3r4p4 1997/04/18, src: v3r4p0 1994/07/10, sys: usg gcc ansi'
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> gap3.load_package("chevie")
>>> gap3.version() # random  # not tested
'lib: v3r4p4 1997/04/18, src: v3r4p0 1994/07/10, sys: usg gcc ansi'
```

Working with GAP3 lists. Note that GAP3 lists are 1-indexed:

```
sage: # optional - gap3
sage: L = gap3([1,2,3])
sage: L[1]
1
sage: L[2]
2
sage: 3 in L
True
sage: 4 in L
False
sage: m = gap3([[1,2],[3,4]])
sage: m[2,1]
3
sage: [1,2] in m
True
sage: [3,2] in m
False
sage: gap3([1,2]) in m
True
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> L = gap3([Integer(1),Integer(2),Integer(3)])
>>> L[Integer(1)]
1
>>> L[Integer(2)]
2
>>> Integer(3) in L
True
>>> Integer(4) in L
False
>>> m = gap3([[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
>>> m[Integer(2),Integer(1)]
3
>>> [Integer(1),Integer(2)] in m
True
```

(continues on next page)

(continued from previous page)

```
>>> [Integer(3),Integer(2)] in m
False
>>> gap3([Integer(1),Integer(2)]) in m
True
```

Controlling variable names used by GAP3:

```
sage: # optional - gap3
sage: gap3('2', name='x')
2
sage: gap3('x')
2
sage: gap3.unbind('x')
sage: gap3('x')
Traceback (most recent call last):
...
TypeError: Gap3 produced error output
Error, Variable: 'x' must have a value
...
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> gap3('2', name='x')
2
>>> gap3('x')
2
>>> gap3.unbind('x')
>>> gap3('x')
Traceback (most recent call last):
...
TypeError: Gap3 produced error output
Error, Variable: 'x' must have a value
...
```

class sage.interfaces.gap3.**GAP3Element**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: `GapElement_generic`

A GAP3 element.

> **Note**
>
> If the corresponding GAP3 element is a GAP3 record, then the class is changed to a `GAP3Record`.

INPUT:

- `parent` – the GAP3 session
- `value` – the GAP3 command as a string
- `is_name` – boolean (default: `False`); if `True`, then `value` is the variable name for the object
- `name` – string (default: `None`); the variable name to use for the object. If `None`, then a variable name is generated

> **Note**
>
> If you pass `E`, `X` or `Z` for `name`, then an error is raised because these are sacred variable names in GAP3 that should never be redefined. Sage raises an error because GAP3 does not!

EXAMPLES:

```
sage: # optional - gap3
sage: from sage.interfaces.gap3 import GAP3Element
sage: gap3 = Gap3()
sage: GAP3Element(gap3, value='3+2')
5
sage: GAP3Element(gap3, value='sage0', is_name=True)
5
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> from sage.interfaces.gap3 import GAP3Element
>>> gap3 = Gap3()
>>> GAP3Element(gap3, value='3+2')
5
>>> GAP3Element(gap3, value='sage0', is_name=True)
5
```

AUTHORS:

- Franco Saliola (Feb 2010)

class `sage.interfaces.gap3.`**`GAP3Record`**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: `GAP3Element`

A GAP3 record.

> **Note**
>
> This class should not be called directly, use GAP3Element instead. If the corresponding GAP3 element is a GAP3 record, then the class is changed to a `GAP3Record`.

AUTHORS:

- Franco Saliola (Feb 2010)

operations()

Return a list of the GAP3 operations for the record.

OUTPUT: list of strings – operations of the record

EXAMPLES:

```
sage: S5 = gap3.SymmetricGroup(5)              #optional - gap3
sage: S5.operations()                          #optional - gap3
[..., 'NormalClosure', 'NormalIntersection', 'Normalizer',
'NumberConjugacyClasses', 'PCore', 'Radical', 'SylowSubgroup',
'TrivialSubgroup', 'FusionConjugacyClasses', 'DerivedSeries', ...]
```

(continues on next page)

(continued from previous page)

```
sage: S5.DerivedSeries()                           #optional - gap3
[ Group( (1,5), (2,5), (3,5), (4,5) ),
  Subgroup( Group( (1,5), (2,5), (3,5), (4,5) ),
            [ (1,2,5), (1,3,5), (1,4,5) ] ) ]
```

```
>>> from sage.all import *
>>> S5 = gap3.SymmetricGroup(Integer(5))                  #optional - gap3
>>> S5.operations()                              #optional - gap3
[..., 'NormalClosure', 'NormalIntersection', 'Normalizer',
'NumberConjugacyClasses', 'PCore', 'Radical', 'SylowSubgroup',
'TrivialSubgroup', 'FusionConjugacyClasses', 'DerivedSeries', ...]
>>> S5.DerivedSeries()                           #optional - gap3
[ Group( (1,5), (2,5), (3,5), (4,5) ),
  Subgroup( Group( (1,5), (2,5), (3,5), (4,5) ),
            [ (1,2,5), (1,3,5), (1,4,5) ] ) ]
```

recfields()

Return a list of the fields for the record. (Record fields are akin to object attributes in Sage.)

OUTPUT: list of strings – the field records

EXAMPLES:

```
sage: S5 = gap3.SymmetricGroup(5)                  #optional - gap3
sage: S5.recfields()                               #optional - gap3
['isDomain', 'isGroup', 'identity', 'generators',
 'operations', 'isPermGroup', 'isFinite', '1', '2',
 '3', '4', 'degree']
sage: S5.degree                                          #optional - gap3
5
```

```
>>> from sage.all import *
>>> S5 = gap3.SymmetricGroup(Integer(5))                  #optional - gap3
>>> S5.recfields()                               #optional - gap3
['isDomain', 'isGroup', 'identity', 'generators',
 'operations', 'isPermGroup', 'isFinite', '1', '2',
 '3', '4', 'degree']
>>> S5.degree                                        #optional - gap3
5
```

class sage.interfaces.gap3.**Gap3**(*command='gap3'*)

Bases: *Gap_generic*

A simple Expect interface to GAP3.

EXAMPLES:

```
sage: from sage.interfaces.gap3 import Gap3
sage: gap3 = Gap3(command='gap3')
```

```
>>> from sage.all import *
>>> from sage.interfaces.gap3 import Gap3
>>> gap3 = Gap3(command='gap3')
```

AUTHORS:

- Franco Saliola (Feb 2010)

console()

Spawn a new GAP3 command-line session.

EXAMPLES:

```
sage: gap3.console()                             # not tested

             ########            Lehrstuhl D fuer Mathematik
           ###    ####           RWTH Aachen
          ##         ##
         ##          #             #######            #########
        ##                        #      ##          ## #     ##
        ##           #           #       ##             #      ##
        ####        ##           ##       #             #      ##
         #####     ###           ##      ##             ##    ##
           ######### #            #########             #######
                     #                                  #
                    ##           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>
```

```
>>> from sage.all import *
>>> gap3.console()                             # not tested

             ########            Lehrstuhl D fuer Mathematik
           ###    ####           RWTH Aachen
          ##         ##
         ##          #             #######            #########
        ##                        #      ##          ## #     ##
        ##           #           #       ##             #      ##
        ####        ##           ##       #             #      ##
         #####     ###           ##      ##             ##    ##
           ######### #            #########             #######
                     #                                  #
                    ##           Version 3              #
                   ###           Release 4.4            #
                  ## #           18 Apr 97              #
                 ##  #
                ##   #  Alice Niemeyer, Werner Nickel,  Martin Schoenert
               ##    #  Johannes Meier, Alex Wegner,    Thomas Bischops
```

(continues on next page)

(continued from previous page)

```
              ##     #  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>
```

cputime (*t=None*)

Return the amount of CPU time that the GAP session has used in seconds.

If `t` is not None, then it returns the difference between the current CPU time and `t`.

EXAMPLES:

```
sage: # optional - gap3
sage: t = gap3.cputime()
sage: t   # random
0.02
sage: gap3.SymmetricGroup(5).Size()
120
sage: gap3.cputime()   # random
0.14999999999999999
sage: gap3.cputime(t)   # random
0.13
```

```
>>> from sage.all import *
>>> # optional - gap3
>>> t = gap3.cputime()
>>> t   # random
0.02
>>> gap3.SymmetricGroup(Integer(5)).Size()
120
>>> gap3.cputime()   # random
0.14999999999999999
>>> gap3.cputime(t)   # random
0.13
```

help (*topic*, *pager=True*)

Print help on the given topic.

INPUT:

- `topic` – string

EXAMPLES:

```
sage: gap3.help('help', pager=False)          #optional - gap3
Help _______________________________________________________...

This  section describes  together with  the following sectio...
help system.  The help system lets you read the manual inter...
```

```
>>> from sage.all import *
>>> gap3.help('help', pager=False)           #optional - gap3
Help _______________________________________________________...
<BLANKLINE>
This  section describes  together with  the following sectio...
help system.  The help system lets you read the manual inter...
```

```
sage: gap3.help('SymmetricGroup', pager=False) #optional - gap3
no section with this name was found
```

```
>>> from sage.all import *
>>> gap3.help('SymmetricGroup', pager=False) #optional - gap3
no section with this name was found
```

sage.interfaces.gap3.**gap3_console**()

Spawn a new GAP3 command-line session.

EXAMPLES:

```
sage: gap3.console()                             # not tested

             ########            Lehrstuhl D fuer Mathematik
           ###    ####           RWTH Aachen
          ##         ##
         ##          #             #######            #########
        ##                        #      ##          ## #     ##
        ##           #           #       ##             #      ##
        ####        ##           ##       #             #      ##
         #####     ###           ##      ##             ##    ##
           ######### #            #########             #######
                     #                                  #
                    ##           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>
```

```
>>> from sage.all import *
>>> gap3.console()                             # not tested

             ########            Lehrstuhl D fuer Mathematik
           ###    ####           RWTH Aachen
          ##         ##
         ##          #             #######            #########
```

(continues on next page)

(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'
```

```
>>> from sage.all import *
>>> gap3_version()                                          # random, optional -␣
↪gap3
'lib: v3r4p4 1997/04/18, src: v3r4p0 1994/07/10, sys: usg gcc ansi'
```

CHAPTER

TWELVE

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.

CHAPTER

THIRTEEN

PEXPECT INTERFACE TO GIAC

(You should prefer the cython interface: giacpy_sage and its libgiac command)

(adapted by F. Han from William Stein and Gregg Musiker maple's interface)

You must have the Giac interpreter installed and available as the command `giac` in your PATH in order to use this interface. You need a giac version supporting "giac –sage" (roughly after 0.9.1). In this case you do not have to install any optional Sage packages. If giac is not already installed, you can download binaries or sources or spkg (follow the sources link) from the homepage:

Homepage <https://www-fourier.ujf-grenoble.fr/~parisse/giac.html>

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

If the giac spkg is installed, you should find the full html documentation there:

```
$SAGE_LOCAL/share/giac/doc/en/cascmd_local/index.html
```

EXAMPLES:

```
sage: giac('3 * 5')
15
sage: giac.eval('ifactor(2005)')
'5*401'
sage: giac.ifactor(2005)
2005
sage: l=giac.ifactors(2005) ; l; l[2]
[5,1,401,1]
401
sage: giac.fsolve('x^2=cos(x)+4', 'x','0..5')
[1.9140206190...
sage: giac.factor('x^4 - y^4')
(x-y)*(x+y)*(x^2+y^2)
sage: R.<x,y>=QQ[];f=(x+y)^5;f2=giac(f);(f-f2).normal()
0
sage: x,y=giac('x,y'); giac.int(y/(cos(2*x)+cos(x)),x)     # random
y*2*((-(tan(x/2)))/6+(-2*1/6/sqrt(3))*ln(abs(6*tan(x/2)-2*sqrt(3))/abs(6*tan(x/
↪2)+2*sqrt(3))))
```

```
>>> from sage.all import *
>>> giac('3 * 5')
15
```

(continues on next page)

(continued from previous page)

```
>>> giac.eval('ifactor(2005)')
'5*401'
>>> giac.ifactor(Integer(2005))
2005
>>> l=giac.ifactors(Integer(2005)) ; l; l[Integer(2)]
[5,1,401,1]
401
>>> giac.fsolve('x^2=cos(x)+4', 'x','0..5')
[1.9140206190...
>>> giac.factor('x^4 - y^4')
(x-y)*(x+y)*(x^2+y^2)
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2);f=(x+y)**Integer(5);f2=giac(f);(f-f2).
→normal()
0
>>> x,y=giac('x,y'); giac.int(y/(cos(Integer(2)*x)+cos(x)),x)      # random
y*2*((-(tan(x/2)))/6+(-2*1/6/sqrt(3))*ln(abs(6*tan(x/2)-2*sqrt(3))/abs(6*tan(x/
→2)+2*sqrt(3))))
```

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

13.1 Tutorial

AUTHORS:

- Gregg Musiker (2006-02-02): initial version.
- Frederic Han: adapted to giac.
- Marcelo Forets (2017-04-06): conversions and cleanup.

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

13.1.1 Syntax

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

1. If you have a giac expression such as

```
factor( (x^4-1));
```

We can write that in sage as

```
sage: giac('factor(x^4-1)')
(x-1)*(x+1)*(x^2+1)
```

```
>>> from sage.all import *
>>> giac('factor(x^4-1)')
(x-1)*(x+1)*(x^2+1)
```

Notice, there is no need to use a semicolon.

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

```
sage: giac('(x^4-1)').factor()
(x-1)*(x+1)*(x^2+1)
```

```
>>> from sage.all import *
>>> giac('(x^4-1)').factor()
(x-1)*(x+1)*(x^2+1)
```

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

```
sage: giac('(x^12-1)/(x-1)').normal()
x^11+x^10+x^9+x^8+x^7+x^6+x^5+x^4+x^3+x^2+x+1
```

```
>>> from sage.all import *
>>> giac('(x^12-1)/(x-1)').normal()
x^11+x^10+x^9+x^8+x^7+x^6+x^5+x^4+x^3+x^2+x+1
```

13.1.2 Some typical input

The normal command will reduce a rational function to the lowest terms. In giac, simplify is slower than normal because it tries more sophisticated simplifications (ex algebraic extensions) The factor command will factor a polynomial with rational coefficients into irreducible factors over the ring of integers (if your default configuration of giac (cf .xcasrc) has not allowed square roots). So for example,

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

```
>>> from sage.all import *
>>> giac('(x^12-1)').factor( )
(x-1)*(x+1)*(x^2+1)*(x^2-x+1)*(x^2+x+1)*(x^4-x^2+1)
```

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

```
>>> from sage.all import *
>>> giac('(x^28-1)').factor( )
(x-1)*(x+1)*(x^2+1)*(x^6-x^5+x^4-x^3+x^2-x+1)*(x^6+x^5+x^4+x^3+x^2+x+1)*(x^12-x^10+x^
↪8-x^6+x^4-x^2+1)
```

13.1.3 Giac console

Another important feature of giac 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 giac console by the command

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

```
>>> from sage.all import *
>>> giac.console()                       # not tested
>>> !giac                                # not tested
```

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

For example, for help on the giac command factors, we type

```
sage: giac.help('factors')                          # not tested
```

```
>>> from sage.all import *
>>> giac.help('factors')                          # not tested
```

```
sage: alpha = giac((1+sqrt(5))/2)
sage: beta = giac(1-sqrt(5))/2
sage: f19  = alpha^19 - beta^19/sqrt(5)
sage: f19
(sqrt(5)/2+1/2)^19-((-sqrt(5)+1)/2)^19/sqrt(5)
sage: (f19-(5778*sqrt(5)+33825)/5).normal()
0
```

```
>>> from sage.all import *
>>> alpha = giac((Integer(1)+sqrt(Integer(5)))/Integer(2))
>>> beta = giac(Integer(1)-sqrt(Integer(5)))/Integer(2)
>>> f19  = alpha**Integer(19) - beta**Integer(19)/sqrt(Integer(5))
>>> f19
(sqrt(5)/2+1/2)^19-((-sqrt(5)+1)/2)^19/sqrt(5)
>>> (f19-(Integer(5778)*sqrt(Integer(5))+Integer(33825))/Integer(5)).normal()
0
```

13.1.4 Function definitions

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

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

In Sage, we write

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

```
>>> from sage.all import *
>>> mysqcu = giac('proc(x) if x > 0 then x^2 else x^3 fi end')
>>> mysqcu(Integer(5))
25
```

(continues on next page)

(continued from previous page)

```
>>> mysqcu(-Integer(5))
-125
```

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

13.1.5 Conversions

The `GiacElement.sage()` method tries to convert a Giac 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 Giac's
- symbolic variables whose names don't pathologically overlap with objects already defined in Sage.

This method will not work when Giac's output includes functions unknown to Sage.

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

```
sage: g = giac('NewFn(x)')
sage: g._sage_(locals={('NewFn', 1): sin})
sin(x)
```

```
>>> from sage.all import *
>>> g = giac('NewFn(x)')
>>> g._sage_(locals={('NewFn', Integer(1)): sin})
sin(x)
```

Moreover, new conversions can be permanently added using Pynac's `register_symbol`, and this is the recommended approach for library code. For more details, see the documentation for `._sage_()`.

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

Bases: *Expect*

Interface to the Giac interpreter.

You must have the optional Giac interpreter installed and available as the command `giac` in your PATH in order to use this interface. Try the command: print(giac._install_hints()) for more informations on giac installation.

Type `giac.[tab]` for a list of all the functions available from your Giac install. Type `giac.[tab]?` for Giac's help about a given function. Type `giac(...)` to create a new Giac object.

Full html documentation for giac is available from your giac installation at `$PREFIX`/share/giac/doc/en/cascmd_en/index.html

EXAMPLES:

Any Giac instruction can be evaluated as a string by the giac command. You can access the giac functions by adding the `giac.` prefix to the usual Giac name.

```
sage: l=giac('normal((y+sqrt(2))^4)'); l
y^4+4*sqrt(2)*y^3+12*y^2+8*sqrt(2)*y+4
sage: f=giac('(u,v)->{ if (u<v){ [u,v] } else { [v,u] }}');f(1,2),f(3,1)
([1,2], [1,3])
```

```
>>> from sage.all import *
>>> l=giac('normal((y+sqrt(2))^4)'); l
y^4+4*sqrt(2)*y^3+12*y^2+8*sqrt(2)*y+4
>>> f=giac('(u,v)->{ if (u<v){ [u,v] } else { [v,u] }}');f(Integer(1),Integer(2)),
↪f(Integer(3),Integer(1))
([1,2], [1,3])
```

The output of the giac command is a Giac object, and it can be used for another giac command.

```
sage: l.factors()
[y+sqrt(2),4]
sage: giac('(x^12-1)').factor( )
(x-1)*(x+1)*(x^2+1)*(x^2-x+1)*(x^2+x+1)*(x^4-x^2+1)
sage: giac('assume(y>0)'); giac('y^2=3').solve('y')
y
...[sqrt(3)]
```

```
>>> from sage.all import *
>>> l.factors()
[y+sqrt(2),4]
>>> giac('(x^12-1)').factor( )
(x-1)*(x+1)*(x^2+1)*(x^2-x+1)*(x^2+x+1)*(x^4-x^2+1)
>>> giac('assume(y>0)'); giac('y^2=3').solve('y')
y
...[sqrt(3)]
```

You can create some Giac elements and avoid many quotes like this:

```
sage: x,y,z=giac('x,y,z');type(y)
<class 'sage.interfaces.giac.GiacElement'>
sage: I1=(1/(cos(2*y)+cos(y))).integral(y,0,pi/4).simplify()
sage: (I1-((-2*ln((sqrt(3)-3*tan(1/8*pi))/(sqrt(3)+3*tan(1/8*pi)))*sqrt(3)-
↪3*tan(1/8*pi))/9)).normal()
0
sage: ((y+z*sqrt(5))*(y-sqrt(5)*z)).normal()
y^2-5*z^2
```

```
>>> from sage.all import *
>>> x,y,z=giac('x,y,z');type(y)
<class 'sage.interfaces.giac.GiacElement'>
>>> I1=(Integer(1)/(cos(Integer(2)*y)+cos(y))).integral(y,Integer(0),pi/
↪Integer(4)).simplify()
>>> (I1-((-Integer(2)*ln((sqrt(Integer(3))-Integer(3)*tan(Integer(1)/
↪Integer(8)*pi))/(sqrt(Integer(3))+Integer(3)*tan(Integer(1)/
↪Integer(8)*pi)))*sqrt(Integer(3))-Integer(3)*tan(Integer(1)/Integer(8)*pi))/
↪Integer(9))).normal()
0
>>> ((y+z*sqrt(Integer(5)))*(y-sqrt(Integer(5))*z)).normal()
y^2-5*z^2
```

Polynomials or elements of SR can be evaluated directly by the giac interface.

```
sage: R.<a,b> = QQ[]; f = (2+a+b)
sage: p = giac.gcd(f^3+5*f^5,f^2+f^5); p; R(p.sage())
sageVARa^2+2*sageVARa*sageVARb+4*sageVARa+sageVARb^2+4*sageVARb+4
a^2 + 2*a*b + b^2 + 4*a + 4*b + 4
```

```
>>> from sage.all import *
>>> R = QQ['a, b']; (a, b,) = R._first_ngens(2); f = (Integer(2)+a+b)
>>> p = giac.gcd(f**Integer(3)+Integer(5)*f**Integer(5),
↪f**Integer(2)+f**Integer(5)); p; R(p.sage())
sageVARa^2+2*sageVARa*sageVARb+4*sageVARa+sageVARb^2+4*sageVARb+4
a^2 + 2*a*b + b^2 + 4*a + 4*b + 4
```

Variable names in python and giac are independent:

```
sage: a=sqrt(2);giac('Digits:=30;a:=5');a,giac('a'),giac(a),giac(a).evalf()
30
(sqrt(2), 5, sqrt(2), 1.41421356237309504880168872421)
```

```
>>> from sage.all import *
>>> a=sqrt(Integer(2));giac('Digits:=30;a:=5');a,giac('a'),giac(a),giac(a).evalf()
30
(sqrt(2), 5, sqrt(2), 1.41421356237309504880168872421)
```

clear(*var*)

Clear the variable named var.

EXAMPLES:

```
sage: giac.set('xx', '2')
sage: giac.get('xx')
'2'
sage: giac.clear('xx')
sage: giac.get('xx')
'xx'
```

```
>>> from sage.all import *
>>> giac.set('xx', '2')
>>> giac.get('xx')
'2'
>>> giac.clear('xx')
>>> giac.get('xx')
'xx'
```

completions(*s*)

Return all commands that complete the command starting with the string s.

EXAMPLES:

```
sage: c = giac.completions('cas')
sage: 'cas_setup' in c
True
```

```
>>> from sage.all import *
>>> c = giac.completions('cas')
>>> 'cas_setup' in c
True
```

console()

Spawn a new Giac command-line session.

EXAMPLES:

```
sage: giac_console()                     # not tested - giac
...
Homepage http://www-fourier.ujf-grenoble.fr/~parisse/giac.html
Released under the GPL license 3.0 or above
See http://www.gnu.org for license details
-------------------------------------------------
Press CTRL and D simultaneously to finish session
Type ?commandname for help
0>>
```

```
>>> from sage.all import *
>>> giac_console()                     # not tested - giac
...
Homepage http://www-fourier.ujf-grenoble.fr/~parisse/giac.html
Released under the GPL license 3.0 or above
See http://www.gnu.org for license details
-------------------------------------------------
Press CTRL and D simultaneously to finish session
Type ?commandname for help
0>>
```

cputime(*t=None*)

Return the amount of CPU time that the Giac session has used.

If `t` is not None, then it returns the difference between the current CPU time and `t`.

EXAMPLES:

```
sage: t = giac.cputime()
sage: t                     # random
0.02
sage: x = giac('x')
sage: giac.diff(x^2, x)
2*x
sage: giac.cputime(t)       # random
0.0
```

```
>>> from sage.all import *
>>> t = giac.cputime()
>>> t                     # random
0.02
>>> x = giac('x')
>>> giac.diff(x**Integer(2), x)
2*x
```

(continues on next page)

(continued from previous page)

```
>>> giac.cputime(t)        # random
0.0
```

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

Send the code x to the Giac interpreter. Remark: To enable multi-lines codes in the notebook magic mode: `%giac`, the `\n` are removed before sending the code to giac.

INPUT:

- `code` – str
- `strip` – default is `True` and removes `\n`

EXAMPLES:

```
sage: giac.eval("2+2;\n3")
'4,3'
sage: giac.eval("2+2;\n3",False)
'4\n3'
sage: s='g(x):={\nx+1;\nx+2;\n}'
sage: giac(s)
...x+1...x+2...
sage: giac.g(5)
7
```

```
>>> from sage.all import *
>>> giac.eval("2+2;\n3")
'4,3'
>>> giac.eval("2+2;\n3",False)
'4\n3'
>>> s='g(x):={\nx+1;\nx+2;\n}'
>>> giac(s)
...x+1...x+2...
>>> giac.g(Integer(5))
7
```

expect()

Return the pexpect object for this Giac session.

EXAMPLES:

```
sage: m = Giac()
sage: m.expect() is None
True
sage: m._start()
sage: m.expect()
Giac with PID ... running .../giac --sage
sage: m.quit()
```

```
>>> from sage.all import *
>>> m = Giac()
>>> m.expect() is None
True
>>> m._start()
```

(continues on next page)

(continued from previous page)

```
>>> m.expect()
Giac with PID ... running .../giac --sage
>>> m.quit()
```

get(*var*)

Get the value of the variable var.

EXAMPLES:

```
sage: giac.set('xx', '2')
sage: giac.get('xx')
'2'
```

```
>>> from sage.all import *
>>> giac.set('xx', '2')
>>> giac.get('xx')
'2'
```

help(*string*)

Display Giac help about string.

This is the same as typing "?string" in the Giac console.

INPUT:

- `string` – string to search for in the giac help system

EXAMPLES:

```
sage: giac.help('Psi')          # not tested - depends of giac and $LANG
Psi(a,n)=nth-derivative of the function DiGamma (=ln@Gamma) at point a (Psi(a,
↪0)=Psi(a))...
```

```
>>> from sage.all import *
>>> giac.help('Psi')          # not tested - depends of giac and $LANG
Psi(a,n)=nth-derivative of the function DiGamma (=ln@Gamma) at point a (Psi(a,
↪0)=Psi(a))...
```

set(*var*, *value*)

Set the variable var to the given value.

EXAMPLES:

```
sage: giac.set('xx', '2')
sage: giac.get('xx')
'2'
```

```
>>> from sage.all import *
>>> giac.set('xx', '2')
>>> giac.get('xx')
'2'
```

version()

Wrapper for giac's version().

EXAMPLES:

```
sage: giac.version()
"giac...
```

```
>>> from sage.all import *
>>> giac.version()
"giac...
```

class sage.interfaces.giac.**GiacElement**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: *ExpectElement*

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)

This returns the definite integral if xmin is not None, otherwise an indefinite integral.

EXAMPLES:

```
sage: y=giac('y');f=(sin(2*y)/y).integral(y).simplify(); f
Si(2*y)
sage: f.diff(y).simplify()
sin(2*y)/y
```

```
>>> from sage.all import *
>>> y=giac('y');f=(sin(Integer(2)*y)/y).integral(y).simplify(); f
Si(2*y)
>>> f.diff(y).simplify()
sin(2*y)/y
```

```
sage: f = giac('exp(x^2)').integral('x',0,1) ; f
1.46265174...
sage: x,y=giac('x'),giac('y');integrate(cos(x+y),'x=0..pi').simplify()
-2*sin(y)
```

```
>>> from sage.all import *
>>> f = giac('exp(x^2)').integral('x',Integer(0),Integer(1)) ; f
1.46265174...
>>> x,y=giac('x'),giac('y');integrate(cos(x+y),'x=0..pi').simplify()
-2*sin(y)
```

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)

This returns the definite integral if xmin is not None, otherwise an indefinite integral.

EXAMPLES:

```
sage: y=giac('y');f=(sin(2*y)/y).integral(y).simplify(); f
Si(2*y)
sage: f.diff(y).simplify()
sin(2*y)/y
```

```
>>> from sage.all import *
>>> y=giac('y');f=(sin(Integer(2)*y)/y).integral(y).simplify(); f
Si(2*y)
>>> f.diff(y).simplify()
sin(2*y)/y
```

```
sage: f = giac('exp(x^2)').integral('x',0,1) ; f
1.46265174...
sage: x,y=giac('x'),giac('y');integrate(cos(x+y),'x=0..pi').simplify()
-2*sin(y)
```

```
>>> from sage.all import *
>>> f = giac('exp(x^2)').integral('x',Integer(0),Integer(1)) ; f
1.46265174...
>>> x,y=giac('x'),giac('y');integrate(cos(x+y),'x=0..pi').simplify()
-2*sin(y)
```

sum(*var*, *min=None*, *max=None*)

Return the sum of self with respect to the variable x.

INPUT:

- var – variable
- min – (default: None)
- max – (default: None)

This returns the definite integral if xmin is not None, otherwise an indefinite integral.

EXAMPLES:

```
sage: giac('1/(1+k^2)').sum('k',-oo,+infinity).simplify()
(pi*exp(pi)^2+pi)/(exp(pi)^2-1)
```

```
>>> from sage.all import *
>>> giac('1/(1+k^2)').sum('k',-oo,+infinity).simplify()
(pi*exp(pi)^2+pi)/(exp(pi)^2-1)
```

unapply(*var*)

Create a Giac function in the given arguments from a Giac symbol.

EXAMPLES:

```
sage: f=giac('y^3+1+t')
sage: g=(f.unapply('y,t'))
sage: g
(y,t)->y^3+1+t
sage: g(1,2)
4
```

```
>>> from sage.all import *
>>> f=giac('y^3+1+t')
>>> g=(f.unapply('y,t'))
>>> g
(y,t)->y^3+1+t
>>> g(Integer(1),Integer(2))
4
```

class sage.interfaces.giac.**GiacFunction**(*parent*, *name*)

Bases: *ExpectFunction*

class sage.interfaces.giac.**GiacFunctionElement**(*obj*, *name*)

Bases: *FunctionElement*

sage.interfaces.giac.**giac_console**()

Spawn a new Giac command-line session.

EXAMPLES:

```
sage: giac.console()  # not tested - giac
...
Homepage http://www-fourier.ujf-grenoble.fr/~parisse/giac.html
Released under the GPL license 3.0 or above
See http://www.gnu.org for license details
-------------------------------------------------
Press CTRL and D simultaneously to finish session
Type ?commandname for help
```

```
>>> from sage.all import *
>>> giac.console()  # not tested - giac
...
Homepage http://www-fourier.ujf-grenoble.fr/~parisse/giac.html
Released under the GPL license 3.0 or above
See http://www.gnu.org for license details
-------------------------------------------------
Press CTRL and D simultaneously to finish session
Type ?commandname for help
```

sage.interfaces.giac.**reduce_load_Giac**()

Return the giac object created in sage.interfaces.giac.

EXAMPLES:

```
sage: from sage.interfaces.giac import reduce_load_Giac
sage: reduce_load_Giac()
Giac
```

```
>>> from sage.all import *
>>> from sage.interfaces.giac import reduce_load_Giac
>>> reduce_load_Giac()
Giac
```

CHAPTER

FOURTEEN

INTERFACE TO THE GNUPLOT INTERPRETER

class sage.interfaces.gnuplot.**Gnuplot**

Bases: SageObject

Interface to the Gnuplot interpreter.

console()

gnuplot()

interact(*cmd*)

plot(*cmd*, *file=None*, *verbose=True*, *reset=True*)

Draw the plot described by cmd, and possibly also save to an eps or png file.

INPUT:

- cmd – string
- file – string (default: None); if specified save plot to given file, which may be either an eps (default) or png file
- verbose – print some info
- reset – true; reset gnuplot before making graph

OUTPUT: displays graph

Note

Note that ^ s are replaced by ** s before being passed to gnuplot.

plot3d(*f*, *xmin=-1*, *xmax=1*, *ymin=-1*, *ymax=1*, *zmin=-1*, *zmax=1*, *title=None*, *samples=25*, *isosamples=20*, *xlabel='x'*, *ylabel='y'*, *interact=True*)

plot3d_parametric(*f='cos(u)*(3 + v*cos(u/2)), sin(u)*(3 + v*cos(u/2)), v*sin(u/2)'*, *range1='[u=-pi:pi]'*, *range2='[v=-0.2:0.2]'*, *samples=50*, *title=None*, *interact=True*)

Draw a parametric 3d surface and rotate it interactively.

INPUT:

- f – string; a function of two variables, e.g., 'cos(u)*(3 + v*cos(u/2)), sin(u)*(3 + v*cos(u/2)), v*sin(u/2)'
- range1 – string; range of values for one variable, e.g., '[u=-pi:pi]'
- range2 – string; range of values for another variable, e.g., '[v=-0.2:0.2]'

- `samples` – integer; number of sample points to use
- `title` – string; title of the graph

EXAMPLES:

```
sage: gnuplot.plot3d_parametric('v^2*sin(u), v*cos(u), v*(1-v)')   # optional␣
↪- gnuplot, not tested (since something pops up)
```

```
>>> from sage.all import *
>>> gnuplot.plot3d_parametric('v^2*sin(u), v*cos(u), v*(1-v)')   # optional -␣
↪gnuplot, not tested (since something pops up)
```

sage.interfaces.gnuplot.**gnuplot_console**()

CHAPTER

FIFTEEN

INTERFACE TO THE GP CALCULATOR OF PARI/GP

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

EXAMPLES: We illustrate objects that wrap GP objects (gp is the PARI interpreter):

```
sage: M = gp('[1,2;3,4]')
sage: M
[1, 2; 3, 4]
sage: M * M
[7, 10; 15, 22]
sage: M + M
[2, 4; 6, 8]
sage: M.matdet()
-2
```

```
>>> from sage.all import *
>>> M = gp('[1,2;3,4]')
>>> M
[1, 2; 3, 4]
>>> M * M
[7, 10; 15, 22]
>>> M + M
[2, 4; 6, 8]
>>> M.matdet()
-2
```

```
sage: E = gp.ellinit([1,2,3,4,5])
sage: E.ellglobalred()
[10351, [1, -1, 0, -1], 1, [11, 1; 941, 1], [[1, 5, 0, 1], [1, 5, 0, 1]]]
sage: E.ellan(20)
[1, 1, 0, -1, -3, 0, -1, -3, -3, -3, -1, 0, 1, -1, 0, -1, 5, -3, 4, 3]
```

```
>>> from sage.all import *
>>> E = gp.ellinit([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)])
>>> E.ellglobalred()
[10351, [1, -1, 0, -1], 1, [11, 1; 941, 1], [[1, 5, 0, 1], [1, 5, 0, 1]]]
>>> E.ellan(Integer(20))
[1, 1, 0, -1, -3, 0, -1, -3, -3, -3, -1, 0, 1, -1, 0, -1, 5, -3, 4, 3]
```

```
sage: primitive_root(7)
3
sage: x = gp("znlog( Mod(2,7), Mod(3,7))")
sage: 3^x % 7
2
```

```
>>> from sage.all import *
>>> primitive_root(Integer(7))
3
>>> x = gp("znlog( Mod(2,7), Mod(3,7))")
>>> Integer(3)**x % Integer(7)
2
```

```
sage: print(gp("taylor(sin(x),x)"))
x - 1/6*x^3 + 1/120*x^5 - 1/5040*x^7 + 1/362880*x^9 - 1/39916800*x^11 + 1/
↪6227020800*x^13 - 1/1307674368000*x^15 + O(x^16)
```

```
>>> from sage.all import *
>>> print(gp("taylor(sin(x),x)"))
x - 1/6*x^3 + 1/120*x^5 - 1/5040*x^7 + 1/362880*x^9 - 1/39916800*x^11 + 1/
↪6227020800*x^13 - 1/1307674368000*x^15 + O(x^16)
```

GP has a powerful very efficient algorithm for numerical computation of integrals.

```
sage: gp("a = intnum(x=0,6,sin(x))")
0.039829713349633979454347702077075594548
sage: gp("a")
0.039829713349633979454347702077075594548
sage: gp.kill("a")
sage: gp("a")
a
```

```
>>> from sage.all import *
>>> gp("a = intnum(x=0,6,sin(x))")
0.039829713349633979454347702077075594548
>>> gp("a")
0.039829713349633979454347702077075594548
>>> gp.kill("a")
>>> gp("a")
a
```

Note that gp ASCII plots *do* work in Sage, as follows:

```
sage: print(gp.eval("plot(x=0,6,sin(x))"))

0.9988963 |''''''''''''_x...x_'''''''''''''''''''''''''''''''''''''''''''|
          |          x"        "x                                        |
          |        _"            "_                                      |
          |       x                x                                     |
          |      "                  "                                    |
          |     "                    "                                   |
          |   _"                      "_                                 |
```

(continues on next page)

(continued from previous page)

```
          |  _                          _                               |
          | _                            _                              |
          |_                              _                             |
          _                                                             |
          `````````````````````````````````"``````````````````````````````
          | " |
          | " |
          | "   "
          | "_   _"|
          | _   _ |
          | _   _ |
          | x   x |
          | "_   _" |
          | x_   _x |
-0.998955 |..."x____x".........|
          0   6
```

```
>>> from sage.all import *
>>> print(gp.eval("plot(x=0,6,sin(x))"))
<BLANKLINE>
0.9988963 |''''''''''''_x...x_''|
          | x"   "x |
          | _"   "_ |
          | x   x |
          | "   " |
          | "   " |
          | _"   "_ |
          | _   _ |
          | _   _ |
          |_   _ |
          _ |
          `````````````````````````````````"``````````````````````````````
          | " |
          | " |
          | "   "
          | "_   _"|
          | _   _ |
          | _   _ |
          | x   x |
          | "_   _" |
          | x_   _x |
-0.998955 |..."x____x".........|
          0   6
```

The GP 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"'%10^10000   # ten thousand character string.
sage: a = gp.eval(t)
sage: a = gp(t)
```

```
>>> from sage.all import *
>>> t = '"%s"'%Integer(10)**Integer(10000)   # ten thousand character string.
```

(continues on next page)

(continued from previous page)

```
>>> a = gp.eval(t)
>>> a = gp(t)
```

In Sage, the PARI large Galois groups datafiles should be installed by default:

```
sage: f = gp('x^9 - x - 2')
sage: f.polgalois()
[362880, -1, 34, "S9"]
```

```
>>> from sage.all import *
>>> f = gp('x^9 - x - 2')
>>> f.polgalois()
[362880, -1, 34, "S9"]
```

AUTHORS:

- William Stein
- David Joyner: some examples
- William Stein (2006-03-01): added tab completion for methods: gp.[tab] and x = gp(blah); x.[tab]
- William Stein (2006-03-01): updated to work with PARI 2.2.12-beta
- William Stein (2006-05-17): updated to work with PARI 2.2.13-beta

class `sage.interfaces.gp.`**Gp**(*stacksize=10000000*, *maxread=None*, *script_subdirectory=None*, *logfile=None*, *server=None*, *server_tmpdir=None*, *init_list_length=1024*, *seed=None*)

Bases: `ExtraTabCompletion`, *Expect*

Interface to the PARI gp interpreter.

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

INPUT:

- `stacksize` – integer (default: 10000000); the initial PARI stacksize in bytes (default: 10MB)
- `script_subdirectory` – string (default: `None`); name of the subdirectory of `SAGE_EXTCODE/pari` from which to read scripts
- `logfile` – string (default: `None`); log file for the pexpect interface
- `server` – name of remote server
- `server_tmpdir` – name of temporary directory on remote server
- `init_list_length` – integer (default: 1024); length of initial list of local variables
- `seed` – integer (default: random); random number generator seed for pari

EXAMPLES:

```
sage: Gp()
PARI/GP interpreter
```

```
>>> from sage.all import *
>>> Gp()
PARI/GP interpreter
```

console()

Spawn a new GP command-line session.

EXAMPLES:

```
sage: gp.console()  # not tested
GP/PARI CALCULATOR Version 2.4.3 (development svn-12577)
amd64 running linux (x86-64/GMP-4.2.1 kernel) 64-bit version
compiled: Jul 21 2010, gcc-4.6.0 20100705 (experimental) (GCC)
(readline v6.0 enabled, extended help enabled)
```

```
>>> from sage.all import *
>>> gp.console()  # not tested
GP/PARI CALCULATOR Version 2.4.3 (development svn-12577)
amd64 running linux (x86-64/GMP-4.2.1 kernel) 64-bit version
compiled: Jul 21 2010, gcc-4.6.0 20100705 (experimental) (GCC)
(readline v6.0 enabled, extended help enabled)
```

cputime(*t=None*)

cputime for pari - cputime since the pari process was started.

INPUT:

- `t` – (default: `None`) if not None, then returns time since t

> **Warning**
>
> If you call gettime explicitly, e.g., gp.eval('gettime'), you will throw off this clock.

EXAMPLES:

```
sage: gp.cputime()          # random output
0.008000000000000002
sage: gp.factor('2^157-1')
[852133201, 1; 60726444167, 1; 1654058017289, 1; 2134387368610417, 1]
sage: gp.cputime()          # random output
0.26900000000000002
```

```
>>> from sage.all import *
>>> gp.cputime()          # random output
0.008000000000000002
>>> gp.factor('2^157-1')
[852133201, 1; 60726444167, 1; 1654058017289, 1; 2134387368610417, 1]
>>> gp.cputime()          # random output
0.26900000000000002
```

get(*var*)

Get the value of the GP variable var.

INPUT:

- `var` – string; a valid GP variable identifier

EXAMPLES:

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

```
>>> from sage.all import *
>>> gp.set('x', '2')
>>> gp.get('x')
'2'
```

get_default(*var*)

Return the current value of a PARI gp configuration variable.

INPUT:

- `var` – string; the name of a PARI gp configuration variable (see `gp.default()` for a list)

OUTPUT: string; the value of the variable

EXAMPLES:

```
sage: gp.get_default('log')
0
sage: gp.get_default('datadir')
'.../share/pari'
sage: gp.get_default('seriesprecision')
16
sage: gp.get_default('realprecision')
38
```

```
>>> from sage.all import *
>>> gp.get_default('log')
0
>>> gp.get_default('datadir')
'.../share/pari'
>>> gp.get_default('seriesprecision')
16
>>> gp.get_default('realprecision')
38
```

get_precision()

Return the current PARI precision for real number computations.

EXAMPLES:

```
sage: gp.get_precision()
38
```

```
>>> from sage.all import *
>>> gp.get_precision()
38
```

get_real_precision()

Return the current PARI precision for real number computations.

EXAMPLES:

```
sage: gp.get_precision()
38
```

```
>>> from sage.all import *
>>> gp.get_precision()
38
```

get_series_precision()

Return the current PARI power series precision.

EXAMPLES:

```
sage: gp.get_series_precision()
16
```

```
>>> from sage.all import *
>>> gp.get_series_precision()
16
```

help(*command*)

Return GP's help for `command`.

EXAMPLES:

```
sage: gp.help('gcd')
'gcd(x,{y}): greatest common divisor of x and y.'
```

```
>>> from sage.all import *
>>> gp.help('gcd')
'gcd(x,{y}): greatest common divisor of x and y.'
```

kill(*var*)

Kill the value of the GP variable var.

INPUT:

- `var` – string; a valid GP variable identifier

EXAMPLES:

```
sage: gp.set('xx', '22')
sage: gp.get('xx')
'22'
sage: gp.kill('xx')
sage: gp.get('xx')
'xx'
```

```
>>> from sage.all import *
>>> gp.set('xx', '22')
>>> gp.get('xx')
```

(continues on next page)

(continued from previous page)

```
'22'
>>> gp.kill('xx')
>>> gp.get('xx')
'xx'
```

new_with_bits_prec(*s*, *precision=0*)

Create a GP object from s with `precision` bits of precision. GP actually automatically increases this precision to the nearest word (i.e. the next multiple of 32 on a 32-bit machine, or the next multiple of 64 on a 64-bit machine).

EXAMPLES:

```
sage: # needs sage.symbolic
sage: pi_def = gp(pi); pi_def
3.1415926535897932384626433832795028842
sage: pi_def.precision()
38
sage: pi_150 = gp.new_with_bits_prec(pi, 150)
sage: new_prec = pi_150.precision(); new_prec
48                                              # 32-bit
57                                              # 64-bit
sage: old_prec = gp.set_precision(new_prec); old_prec
38
sage: pi_150
3.14159265358979323846264338327950288419716939938  # 32-bit
3.14159265358979323846264338327950288419716939937510582098  # 64-bit
sage: gp.set_precision(old_prec)
48                                              # 32-bit
57                                              # 64-bit
sage: gp.get_precision()
38
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> pi_def = gp(pi); pi_def
3.1415926535897932384626433832795028842
>>> pi_def.precision()
38
>>> pi_150 = gp.new_with_bits_prec(pi, Integer(150))
>>> new_prec = pi_150.precision(); new_prec
48                                              # 32-bit
57                                              # 64-bit
>>> old_prec = gp.set_precision(new_prec); old_prec
38
>>> pi_150
3.14159265358979323846264338327950288419716939938  # 32-bit
3.14159265358979323846264338327950288419716939937510582098  # 64-bit
>>> gp.set_precision(old_prec)
48                                              # 32-bit
57                                              # 64-bit
>>> gp.get_precision()
38
```

set (*var*, *value*)

Set the GP variable var to the given value.

INPUT:

- `var` – string; a valid GP variable identifier
- `value` – a value for the variable

EXAMPLES:

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

```
>>> from sage.all import *
>>> gp.set('x', '2')
>>> gp.get('x')
'2'
```

set_default (*var*, *value*)

Set a PARI gp configuration variable, and return the old value.

INPUT:

- `var` – string; the name of a PARI gp configuration variable (see `gp.default()` for a list)
- `value` – the value to set the variable to

EXAMPLES:

```
sage: old_prec = gp.set_default('realprecision', 110)
sage: gp.get_default('realprecision')
115
sage: gp.set_default('realprecision', old_prec)
115
sage: gp.get_default('realprecision')
38
```

```
>>> from sage.all import *
>>> old_prec = gp.set_default('realprecision', Integer(110))
>>> gp.get_default('realprecision')
115
>>> gp.set_default('realprecision', old_prec)
115
>>> gp.get_default('realprecision')
38
```

set_precision (*prec*)

Set the PARI precision (in decimal digits) for real computations, and returns the old value.

> **Note**
>
> PARI/GP rounds up precisions to the nearest machine word, so the result of `get_precision()` is not always the same as the last value inputted to `set_precision()`.

EXAMPLES:

```
sage: old_prec = gp.set_precision(53); old_prec
38
sage: gp.get_precision()
57
sage: gp.set_precision(old_prec)
57
sage: gp.get_precision()
38
```

```
>>> from sage.all import *
>>> old_prec = gp.set_precision(Integer(53)); old_prec
38
>>> gp.get_precision()
57
>>> gp.set_precision(old_prec)
57
>>> gp.get_precision()
38
```

set_real_precision(*prec*)

Set the PARI precision (in decimal digits) for real computations, and returns the old value.

> **Note**
>
> PARI/GP rounds up precisions to the nearest machine word, so the result of `get_precision()` is not always the same as the last value inputted to `set_precision()`.

EXAMPLES:

```
sage: old_prec = gp.set_precision(53); old_prec
38
sage: gp.get_precision()
57
sage: gp.set_precision(old_prec)
57
sage: gp.get_precision()
38
```

```
>>> from sage.all import *
>>> old_prec = gp.set_precision(Integer(53)); old_prec
38
>>> gp.get_precision()
57
>>> gp.set_precision(old_prec)
57
>>> gp.get_precision()
38
```

set_seed(*seed=None*)

Set the seed for gp interpreter.

The seed should be an integer.

EXAMPLES:

```
sage: g = Gp()
sage: g.set_seed(1)
1
sage: [g.random() for i in range(5)]
[1546275796, 879788114, 1745191708, 771966234, 1247963869]
```

```
>>> from sage.all import *
>>> g = Gp()
>>> g.set_seed(Integer(1))
1
>>> [g.random() for i in range(Integer(5))]
[1546275796, 879788114, 1745191708, 771966234, 1247963869]
```

set_series_precision(*prec=None*)

Set the PARI power series precision, and returns the old precision.

EXAMPLES:

```
sage: old_prec = gp.set_series_precision(50); old_prec
16
sage: gp.get_series_precision()
50
sage: gp.set_series_precision(old_prec)
50
sage: gp.get_series_precision()
16
```

```
>>> from sage.all import *
>>> old_prec = gp.set_series_precision(Integer(50)); old_prec
16
>>> gp.get_series_precision()
50
>>> gp.set_series_precision(old_prec)
50
>>> gp.get_series_precision()
16
```

version()

Return the version of GP being used.

EXAMPLES:

```
sage: gp.version()  # not tested
((2, 4, 3), 'GP/PARI CALCULATOR Version 2.4.3 (development svn-12577)')
```

```
>>> from sage.all import *
>>> gp.version()  # not tested
((2, 4, 3), 'GP/PARI CALCULATOR Version 2.4.3 (development svn-12577)')
```

class sage.interfaces.gp.**GpElement**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: *ExpectElement*, *GpElement*

EXAMPLES: This example illustrates dumping and loading GP elements to compressed strings.

```
sage: a = gp(39393)
sage: loads(a.dumps()) == a
True
```

```
>>> from sage.all import *
>>> a = gp(Integer(39393))
>>> loads(a.dumps()) == a
True
```

Since dumping and loading uses the string representation of the object, it need not result in an identical object from the point of view of PARI:

```
sage: E = gp('ellinit([1,2,3,4,5])')
sage: loads(dumps(E)) == E
True
sage: x = gp.Pi()/3
sage: loads(dumps(x)) == x
False
sage: x
1.047197551196597746154214461093167628
sage: loads(dumps(x))
1.047197551196597746154214461093167628
```

```
>>> from sage.all import *
>>> E = gp('ellinit([1,2,3,4,5])')
>>> loads(dumps(E)) == E
True
>>> x = gp.Pi()/Integer(3)
>>> loads(dumps(x)) == x
False
>>> x
1.047197551196597746154214461093167628
>>> loads(dumps(x))
1.047197551196597746154214461093167628
```

The two elliptic curves look the same, but internally the floating point numbers are slightly different.

is_string()

Tell whether this element is a string.

EXAMPLES:

```
sage: gp('"abc"').is_string()
True
sage: gp('[1,2,3]').is_string()
False
```

```
>>> from sage.all import *
>>> gp('"abc"').is_string()
True
>>> gp('[1,2,3]').is_string()
False
```

sage.interfaces.gp.**gp_console**()

Spawn a new GP command-line session.

EXAMPLES:

```
sage: gp.console()  # not tested
GP/PARI CALCULATOR Version 2.4.3 (development svn-12577)
amd64 running linux (x86-64/GMP-4.2.1 kernel) 64-bit version
compiled: Jul 21 2010, gcc-4.6.0 20100705 (experimental) (GCC)
(readline v6.0 enabled, extended help enabled)
```

```
>>> from sage.all import *
>>> gp.console()  # not tested
GP/PARI CALCULATOR Version 2.4.3 (development svn-12577)
amd64 running linux (x86-64/GMP-4.2.1 kernel) 64-bit version
compiled: Jul 21 2010, gcc-4.6.0 20100705 (experimental) (GCC)
(readline v6.0 enabled, extended help enabled)
```

sage.interfaces.gp.**gp_version**()

EXAMPLES:

```
sage: gp.version()  # not tested
((2, 4, 3), 'GP/PARI CALCULATOR Version 2.4.3 (development svn-12577)')
```

```
>>> from sage.all import *
>>> gp.version()  # not tested
((2, 4, 3), 'GP/PARI CALCULATOR Version 2.4.3 (development svn-12577)')
```

sage.interfaces.gp.**is_GpElement**(*x*)

Return `True` if `x` is of type *GpElement*.

This function is deprecated; use `isinstance()` (of *sage.interfaces.abc.GpElement*) instead.

EXAMPLES:

```
sage: from sage.interfaces.gp import is_GpElement
sage: is_GpElement(gp(2))
doctest:...: DeprecationWarning: the function is_GpElement is deprecated; use
↪isinstance(x, sage.interfaces.abc.GpElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
True
sage: is_GpElement(2)
False
```

```
>>> from sage.all import *
>>> from sage.interfaces.gp import is_GpElement
>>> is_GpElement(gp(Integer(2)))
doctest:...: DeprecationWarning: the function is_GpElement is deprecated; use
↪isinstance(x, sage.interfaces.abc.GpElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
True
>>> is_GpElement(Integer(2))
False
```

`sage.interfaces.gp.reduce_load_GP()`

Return 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
```

```
>>> from sage.all import *
>>> from sage.interfaces.gp import reduce_load_GP
>>> reduce_load_GP()
PARI/GP interpreter
```

CHAPTER

SIXTEEN

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: `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
- `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
```

```
>>> from sage.all import *
>>> from sage.interfaces.jmoldata import JmolData
>>> JData = JmolData()
>>> 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;
↪"
>>> testfile = tmp_filename(ext="DNA.png")
>>> JData.export_image(targetfile=testfile,datafile=script,image_type="PNG")
↪# optional -- java internet
>>> 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()                                                     #␣
↪needs sage.plot
sage: from tempfile import NamedTemporaryFile
sage: archive = NamedTemporaryFile(suffix='.zip')
sage: D.export_jmol(archive.name)                                            #␣
↪needs sage.plot
sage: archive_native = archive.name
sage: script  = f'set defaultdirectory "f{archive_native}"\n'
sage: script += 'script SCRIPT\n'
sage: with NamedTemporaryFile(suffix='.png') as testfile:    # optional - java,
↪ needs sage.plot
....:     JData.export_image(targetfile=testfile.name,
....:                        datafile=script,
....:                        image_type="PNG")
....:     os.path.exists(testfile.name)
True
```

(continues on next page)

(continued from previous page)

```
sage: archive.close()
```

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

is_jmol_available()

Return `True` if jmol 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_jmol_available())
<... 'bool'>
```

```
>>> from sage.all import *
>>> from sage.interfaces.jmoldata import JmolData
>>> JData = JmolData()
>>> type(JData.is_jmol_available())
<... 'bool'>
```

is_jvm_available()

Return `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'>
```

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

jmolpath()

Return the path to the jar file.

EXAMPLES:

```
sage: from sage.interfaces.jmoldata import JmolData
sage: JData = JmolData()
sage: JData.jmolpath()  # needs jmol
'.../JmolData.jar'
```

```
>>> from sage.all import *
>>> from sage.interfaces.jmoldata import JmolData
>>> JData = JmolData()
>>> JData.jmolpath()  # needs jmol
'.../JmolData.jar'
```

CHAPTER

SEVENTEEN

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 are return the result as a KASH object.
4. `kash.eval(expr)` - Evaluation of arbitrary KASH expressions, with the result returned as a string.

17.1 Issues

For some reason hitting `Control` + `C` to interrupt a calculation does not work correctly. (TODO)

17.2 Tutorial

The examples in this tutorial require that kash be installed.

17.2.1 Basics

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

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

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

```
>>> from sage.all import *
>>> a = kash('(9 - 7) * (5 + 6)'); a              # optional -- kash
22
>>> 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
1124000727777607680000
```

```
>>> from sage.all import *
>>> a*a                                           # optional -- kash
484
>>> a.Factorial()                                 # optional -- kash
1124000727777607680000
```

17.2.2 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.

17.2.3 Arithmetic

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

```
sage: # optional - kash
sage: a = kash(12345)
sage: b = kash(25)
sage: a/b
2469/5
sage: a**b
1937659030411463935651167391656422626577614411586152317674869233464019922771432158872
↪87137603759765625
```

```
>>> from sage.all import *
>>> # optional - kash
>>> a = kash(Integer(12345))
>>> b = kash(Integer(25))
```

(continues on next page)

(continued from previous page)

```
>>> a/b
2469/5
>>> a**b
1937659030411463935651167391656422626577614411586152317674869233464019922771432158872
↪87137603759765625
```

17.2.4 Variable assignment

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

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

```
>>> from sage.all import *
>>> a = kash('32233')                          # optional -- kash
>>> 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"
```

```
>>> from sage.all import *
>>> kash.eval('a')                             # optional -- kash
"Error, the variable 'a' must have a value"
```

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

```
sage: a.name()                                   # somewhat random; optional - kash
'sage0'
sage: kash(a.name())                             # optional -- kash
32233
```

```
>>> from sage.all import *
>>> a.name()                                   # somewhat random; optional - kash
'sage0'
>>> kash(a.name())                             # optional -- kash
32233
```

17.2.5 Integers and Rationals

We illustrate arithmetic with integers and rationals in KASH.

```
sage: # optional - kash
sage: F = kash.Factorization(4352)
sage: F[1]
<2, 8>
sage: F[2]
<17, 1>
```

(continues on next page)

(continued from previous page)

```
sage: F
[ <2, 8>, <17, 1> ], extended by:
  ext1 := 1,
  ext2 := Unassign
```

```
>>> from sage.all import *
>>> # optional - kash
>>> F = kash.Factorization(Integer(4352))
>>> F[Integer(1)]
<2, 8>
>>> F[Integer(2)]
<17, 1>
>>> F
[ <2, 8>, <17, 1> ], extended by:
  ext1 := 1,
  ext2 := Unassign
```

Note

For some very large numbers KASH's integer factorization seems much faster than PARI's (which is the default in Sage).

```
sage: # optional - kash
sage: kash.GCD(15,25)
5
sage: kash.LCM(15,25)
75
sage: kash.Div(25,15)
1
sage: kash(17) % kash(5)
2
sage: kash.IsPrime(10007)
TRUE
sage: kash.IsPrime(2005)
FALSE

sage: kash.NextPrime(10007)                    # optional -- kash
10009
```

```
>>> from sage.all import *
>>> # optional - kash
>>> kash.GCD(Integer(15),Integer(25))
5
>>> kash.LCM(Integer(15),Integer(25))
75
>>> kash.Div(Integer(25),Integer(15))
1
>>> kash(Integer(17)) % kash(Integer(5))
2
>>> kash.IsPrime(Integer(10007))
```

(continues on next page)

(continued from previous page)

```
TRUE
>>> kash.IsPrime(Integer(2005))
FALSE

>>> kash.NextPrime(Integer(10007))                    # optional -- kash
10009
```

17.2.6 Real and Complex Numbers

```
sage: # optional - kash
sage: kash.Precision()
30
sage: kash('R')
Real field of precision 30
sage: kash.Precision(40)
40
sage: kash('R')
Real field of precision 40
sage: z = kash('1 + 2*I')
sage: z
1.000000000000000000000000000000000000000 + 2.
↪000000000000000000000000000000000000000*I
sage: z*z
-3.000000000000000000000000000000000000000 + 4.
↪000000000000000000000000000000000000000*I

sage: kash.Cos('1.24')                        # optional -- kash
0.3247962844387762365776934156973803996992
sage: kash('1.24').Cos()                      # optional -- kash
0.3247962844387762365776934156973803996992

sage: kash.Exp('1.24')                        # optional -- kash
3.455613464762675598057615494121998175400

sage: kash.Precision(30)                      # optional -- kash
30
sage: kash.Log('3+4*I')                       # optional -- kash
1.60943791243410037460075933323 + 0.927295218001612232428512462922*I
sage: kash.Log('I')                           # optional -- kash
1.57079632679489661923132169164*I

sage: kash.Sqrt(4)                            # optional -- kash
2.00000000000000000000000000000
sage: kash.Sqrt(2)                            # optional -- kash
1.41421356237309504880168872421

sage: kash.Floor('9/5')                       # optional -- kash
1
sage: kash.Floor('3/5')                       # optional -- kash
0
```

(continues on next page)

(continued from previous page)

```
sage: x_c = kash('3+I')                              # optional -- kash
sage: x_c.Argument()                                 # optional -- kash
0.32175055439664219340140461435 9
sage: x_c.Imaginary()                                # optional -- kash
1.0000000000000000000000000000
```

```
>>> from sage.all import *
>>> # optional - kash
>>> kash.Precision()
30
>>> kash('R')
Real field of precision 30
>>> kash.Precision(Integer(40))
40
>>> kash('R')
Real field of precision 40
>>> z = kash('1 + 2*I')
>>> z
1.000000000000000000000000000000000000000 + 2.
↪000000000000000000000000000000000000000*I
>>> z*z
-3.000000000000000000000000000000000000000 + 4.
↪000000000000000000000000000000000000000*I

>>> kash.Cos('1.24')                               # optional -- kash
0.32479628443877623657769341569738039969 92
>>> kash('1.24').Cos()                             # optional -- kash
0.32479628443877623657769341569738039969 92

>>> kash.Exp('1.24')                               # optional -- kash
3.4556134647626755980576154941219981754 00

>>> kash.Precision(Integer(30))                             # optional -- kash
30
>>> kash.Log('3+4*I')                              # optional -- kash
1.6094379124341003746007593332 3 + 0.92729521800161223242851246292 2*I
>>> kash.Log('I')                                  # optional -- kash
1.5707963267948966192313216916 4*I

>>> kash.Sqrt(Integer(4))                                   # optional -- kash
2.0000000000000000000000000000
>>> kash.Sqrt(Integer(2))                                   # optional -- kash
1.4142135623730950488016887242 1

>>> kash.Floor('9/5')                              # optional -- kash
1
>>> kash.Floor('3/5')                              # optional -- kash
0

>>> x_c = kash('3+I')                              # optional -- kash
>>> x_c.Argument()                                 # optional -- kash
0.32175055439664219340140461435 9
```

(continues on next page)

(continued from previous page)

```
>>> x_c.Imaginary()                              # optional -- kash
1.000000000000000000000000000000
```

17.2.7 Lists

Note that list appends are completely different in KASH than in Python. Use underscore after the function name for the mutation version.

```
sage: # optional - kash
sage: v = kash([1,2,3]); v
[ 1, 2, 3 ]
sage: v[1]
1
sage: v[3]
3
sage: v.Append([5])
[ 1, 2, 3, 5 ]
sage: v
[ 1, 2, 3 ]
sage: v.Append_([5, 6])
SUCCESS
sage: v
[ 1, 2, 3, 5, 6 ]
sage: v.Add(5)
[ 1, 2, 3, 5, 6, 5 ]
sage: v
[ 1, 2, 3, 5, 6 ]
sage: v.Add_(5)
SUCCESS
sage: v
[ 1, 2, 3, 5, 6, 5 ]
```

```
>>> from sage.all import *
>>> # optional - kash
>>> v = kash([Integer(1),Integer(2),Integer(3)]); v
[ 1, 2, 3 ]
>>> v[Integer(1)]
1
>>> v[Integer(3)]
3
>>> v.Append([Integer(5)])
[ 1, 2, 3, 5 ]
>>> v
[ 1, 2, 3 ]
>>> v.Append_([Integer(5), Integer(6)])
SUCCESS
>>> v
[ 1, 2, 3, 5, 6 ]
>>> v.Add(Integer(5))
[ 1, 2, 3, 5, 6, 5 ]
>>> v
```

(continues on next page)

(continued from previous page)

```
[ 1, 2, 3, 5, 6 ]
>>> v.Add_(Integer(5))
SUCCESS
>>> v
[ 1, 2, 3, 5, 6, 5 ]
```

The `Apply` command applies a function to each element of a list:

```
sage: # optional - kash
sage: L = kash([1,2,3,4])
sage: L.Apply('i -> 3*i')
[ 3, 6, 9, 12 ]
sage: L
[ 1, 2, 3, 4 ]
sage: L.Apply('IsEven')
[ FALSE, TRUE, FALSE, TRUE ]
sage: L
[ 1, 2, 3, 4 ]
```

```
>>> from sage.all import *
>>> # optional - kash
>>> L = kash([Integer(1),Integer(2),Integer(3),Integer(4)])
>>> L.Apply('i -> 3*i')
[ 3, 6, 9, 12 ]
>>> L
[ 1, 2, 3, 4 ]
>>> L.Apply('IsEven')
[ FALSE, TRUE, FALSE, TRUE ]
>>> L
[ 1, 2, 3, 4 ]
```

17.2.8 Ranges

the following are examples of ranges.

```
sage: # optional - kash
sage: L = kash('[1..10]')
sage: L
[ 1 .. 10 ]
sage: L = kash('[2,4..100]')
sage: L
[ 2, 4 .. 100 ]
```

```
>>> from sage.all import *
>>> # optional - kash
>>> L = kash('[1..10]')
>>> L
[ 1 .. 10 ]
>>> L = kash('[2,4..100]')
>>> L
[ 2, 4 .. 100 ]
```

17.2.9 Sequences

17.2.10 Tuples

17.2.11 Polynomials

```
sage: # optional - kash
sage: f = kash('X^3 + X + 1')
sage: f + f
2*X^3 + 2*X + 2
sage: f * f
X^6 + 2*X^4 + 2*X^3 + X^2 + 2*X + 1
sage: f.Evaluate(10)
1011
sage: Qx = kash.PolynomialAlgebra('Q')
sage: Qx.gen(1)**5 + kash('7/3')
sage1.1^5 + 7/3
```

```
>>> from sage.all import *
>>> # optional - kash
>>> f = kash('X^3 + X + 1')
>>> f + f
2*X^3 + 2*X + 2
>>> f * f
X^6 + 2*X^4 + 2*X^3 + X^2 + 2*X + 1
>>> f.Evaluate(Integer(10))
1011
>>> Qx = kash.PolynomialAlgebra('Q')
>>> Qx.gen(Integer(1))**Integer(5) + kash('7/3')
sage1.1^5 + 7/3
```

17.2.12 Number Fields

We create an equation order.

```
sage: f = kash('X^5 + 4*X^4 - 56*X^2 -16*X + 192')     # optional -- kash
sage: OK = f.EquationOrder()                           # optional -- kash
sage: OK                                               # optional -- kash
Equation Order with defining polynomial X^5 + 4*X^4 - 56*X^2 - 16*X + 192 over Z
```

```
>>> from sage.all import *
>>> f = kash('X^5 + 4*X^4 - 56*X^2 -16*X + 192')     # optional -- kash
>>> OK = f.EquationOrder()                           # optional -- kash
>>> OK                                               # optional -- kash
Equation Order with defining polynomial X^5 + 4*X^4 - 56*X^2 - 16*X + 192 over Z
```

```
sage: # optional - kash
sage: f = kash('X^5 + 4*X^4 - 56*X^2 -16*X + 192')
sage: O = f.EquationOrder()
sage: a = O.gen(2)
sage: a
[0, 1, 0, 0, 0]
sage: O.Basis()
```

(continues on next page)

(continued from previous page)

```
[
_NG.1,
_NG.2,
_NG.3,
_NG.4,
_NG.5
]
sage: O.Discriminant()
1364202618880
sage: O.MaximalOrder()
Maximal Order of sage2

sage: O = kash.MaximalOrder('X^3 - 77')                    # optional -- kash
sage: I = O.Ideal(5,[2, 1, 0])                             # optional -- kash
sage: I                    # name sage14 below random; optional -- kash
Ideal of sage14
Two element generators:
[5, 0, 0]
[2, 1, 0]

sage: F = I.Factorisation()                    # optional -- kash
sage: F                    # name sage14 random; optional -- kash
[
<Prime Ideal of sage14
Two element generators:
[5, 0, 0]
[2, 1, 0], 1>
]
```

```
>>> from sage.all import *
>>> # optional - kash
>>> f = kash('X^5 + 4*X^4 - 56*X^2 -16*X + 192')
>>> O = f.EquationOrder()
>>> a = O.gen(Integer(2))
>>> a
[0, 1, 0, 0, 0]
>>> O.Basis()
[
_NG.1,
_NG.2,
_NG.3,
_NG.4,
_NG.5
]
>>> O.Discriminant()
1364202618880
>>> O.MaximalOrder()
Maximal Order of sage2

>>> O = kash.MaximalOrder('X^3 - 77')                    # optional -- kash
>>> I = O.Ideal(Integer(5),[Integer(2), Integer(1), Integer(0)])                    ␣
↪      # optional -- kash
```

(continues on next page)

(continued from previous page)

```
>>> I                    # name sage14 below random; optional -- kash
Ideal of sage14
Two element generators:
[5, 0, 0]
[2, 1, 0]

>>> F = I.Factorisation()                  # optional -- kash
>>> F                    # name sage14 random; optional -- kash
[
<Prime Ideal of sage14
Two element generators:
[5, 0, 0]
[2, 1, 0], 1>
]
```

Determining whether an ideal is principal.

```
sage: I.IsPrincipal()                      # optional -- kash
FALSE, extended by:
ext1 := Unassign
```

```
>>> from sage.all import *
>>> I.IsPrincipal()                      # optional -- kash
FALSE, extended by:
ext1 := Unassign
```

Computation of class groups and unit groups:

```
sage: # optional - kash
sage: f = kash('X^5 + 4*X^4 - 56*X^2 -16*X + 192')
sage: O = kash.EquationOrder(f)
sage: OK = O.MaximalOrder()
sage: OK.ClassGroup()
Abelian Group isomorphic to Z/6
  Defined on 1 generator
  Relations:
  6*sage32.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: sage32 to ids/ord^num: _AA
```

```
>>> from sage.all import *
>>> # optional - kash
>>> f = kash('X^5 + 4*X^4 - 56*X^2 -16*X + 192')
>>> O = kash.EquationOrder(f)
>>> OK = O.MaximalOrder()
>>> OK.ClassGroup()
Abelian Group isomorphic to Z/6
  Defined on 1 generator
  Relations:
  6*sage32.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: sage32 to ids/ord^num: _AA
```

```
sage: U = OK.UnitGroup()                                      # optional -- kash
sage: U        # name sage34 below random; optional -- kash
Abelian Group isomorphic to Z/2 + Z + Z
  Defined on 3 generators
  Relations:
  2*sage34.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: sage34 to ord^num: sage30

sage: kash.Apply('x->%s.ext1(x)'%U.name(), U.Generators().List())     # optional --
↪kash
[ [1, -1, 0, 0, 0], [1, 1, 0, 0, 0], [-1, 0, 0, 0, 0] ]
```

```
>>> from sage.all import *
>>> U = OK.UnitGroup()                                      # optional -- kash
>>> U        # name sage34 below random; optional -- kash
Abelian Group isomorphic to Z/2 + Z + Z
  Defined on 3 generators
  Relations:
  2*sage34.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: sage34 to ord^num: sage30

>>> kash.Apply('x->%s.ext1(x)'%U.name(), U.Generators().List())     # optional -- kash
[ [1, -1, 0, 0, 0], [1, 1, 0, 0, 0], [-1, 0, 0, 0, 0] ]
```

17.2.13 Function Fields

```
sage: # optional - kash
sage: k = kash.FiniteField(25)
sage: kT = k.RationalFunctionField()
sage: kTy = kT.PolynomialAlgebra()
sage: T = kT.gen(1)
sage: y = kTy.gen(1)
sage: f = y**3 + T**4 + 1
```

```
>>> from sage.all import *
>>> # optional - kash
>>> k = kash.FiniteField(Integer(25))
>>> kT = k.RationalFunctionField()
>>> kTy = kT.PolynomialAlgebra()
>>> T = kT.gen(Integer(1))
>>> y = kTy.gen(Integer(1))
>>> f = y**Integer(3) + T**Integer(4) + Integer(1)
```

17.3 Long Input

The KASH interface reads in even very long input (using files) in a robust manner, as long as you are creating a new object.

Note

Using `kash.eval` for long input is much less robust, and is not recommended.

```
sage: a = kash(range(10000))                               # optional -- kash
```

```
>>> from sage.all import *
>>> a = kash(range(Integer(10000)))                            # optional --␣
↪kash
```

Note that KASH seems to not support string or integer literals with more than 1024 digits, which is why the above example uses a list unlike for the other interfaces.

class sage.interfaces.kash.**Kash**(*max_workspace_size=None*, *maxread=None*, *script_subdirectory=None*, *restart_on_ctrlc=True*, *logfile=None*, *server=None*, *server_tmpdir=None*)

Bases: *Expect*

Interface to the Kash interpreter.

AUTHORS:

- William Stein and David Joyner

clear(*var*)

Clear the variable named `var`.

Kash variables have a record structure, so if sage1 is a polynomial ring, sage1.1 will be its indeterminate. This prevents us from easily reusing variables, since sage1.1 might still have references even if sage1 does not.

For now, we don't implement variable clearing to avoid these problems, and instead implement this method with a noop.

console()

eval(*x*, *newlines=False*, *strip=True*, ***kwds*)

Send the code in the string s to the Kash interpreter and return the output as a string.

INPUT:

- `s` – string containing Kash code
- `newlines` – boolean (default: `True`); if `False`, remove all backslash-newlines inserted by the Kash output formatter
- `strip` – ignored

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

EXAMPLES:

```
sage: kash.function_call('ComplexToPolar', [1+I], {'Results' : 1})    #␣
↪optional -- kash
1.414213562373095048801688724
```

```
>>> from sage.all import *
>>> kash.function_call('ComplexToPolar', [Integer(1)+I], {'Results' :␣
↪Integer(1)})    # optional -- kash
1.414213562373095048801688724
```

get(*var*)

Get the value of the variable var.

help(*name=None*)

Return help on KASH commands.

This returns help on all commands with a given name. If name is `None`, return the location of the installed Kash HTML documentation.

EXAMPLES:

```
sage: X = kash.help('IntegerRing')   # random; optional -- kash
1439: IntegerRing() -> <ord^rat>
1440: IntegerRing(<elt-ord^rat> m) -> <res^rat>
1441: IntegerRing(<seq()> Q) -> <res^rat>
1442: IntegerRing(<fld^rat> K) -> <ord^rat>
1443: IntegerRing(<fld^fra> K) -> <ord^num>
1444: IntegerRing(<rng> K) -> <rng>
1445: IntegerRing(<fld^pad> L) -> <ord^pad>
```

```
>>> from sage.all import *
>>> X = kash.help('IntegerRing')   # random; optional -- kash
1439: IntegerRing() -> <ord^rat>
1440: IntegerRing(<elt-ord^rat> m) -> <res^rat>
1441: IntegerRing(<seq()> Q) -> <res^rat>
1442: IntegerRing(<fld^rat> K) -> <ord^rat>
1443: IntegerRing(<fld^fra> K) -> <ord^num>
1444: IntegerRing(<rng> K) -> <rng>
1445: IntegerRing(<fld^pad> L) -> <ord^pad>
```

There is one entry in X for each item found in the documentation for this function: If you type `print(X[0])` you will get help on about the first one, printed nicely to the screen.

AUTHORS:

- Sebastion Pauli (2006-02-04): during Sage coding sprint

help_search(*name*)

set(*var*, *value*)

Set the variable var to the given value.

version()

class sage.interfaces.kash.**KashDocumentation**(*iterable=()*, */*)

Bases: `list`

class sage.interfaces.kash.**KashElement**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: *ExpectElement*

sage.interfaces.kash.**is_KashElement**(*x*)

Return `True` if `x` is of type *KashElement*.

EXAMPLES:

```
sage: from sage.interfaces.kash import is_KashElement
sage: is_KashElement(2)
```

(continues on next page)

(continued from previous page)

```
doctest:...: DeprecationWarning: the function is_KashElement is deprecated; use␣
↪isinstance(x, sage.interfaces.abc.KashElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
sage: is_KashElement(kash(2))  # optional - kash
True
```

```
>>> from sage.all import *
>>> from sage.interfaces.kash import is_KashElement
>>> is_KashElement(Integer(2))
doctest:...: DeprecationWarning: the function is_KashElement is deprecated; use␣
↪isinstance(x, sage.interfaces.abc.KashElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
>>> is_KashElement(kash(Integer(2)))  # optional - kash
True
```

sage.interfaces.kash.**kash_console**()

sage.interfaces.kash.**kash_version**()

sage.interfaces.kash.**reduce_load_Kash**()

CHAPTER

EIGHTEEN

LIBRARY INTERFACE TO KENZO

Kenzo is a set of lisp functions to compute homology and homotopy groups of topological spaces.

AUTHORS:

- Miguel Marco, Ana Romero (2019-01): Initial version

For this interface, Kenzo is loaded into ECL which is itself loaded as a C library in Sage. Kenzo objects in this interface are nothing but wrappers around ECL objects.

`sage.interfaces.kenzo.`**`BicomplexSpectralSequence`**(*l*)

Construct the spectral sequence associated to the bicomplex given by a list of morphisms.

INPUT:

- `l` – list of morphisms of chain complexes

OUTPUT: a *`KenzoSpectralSequence`*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import BicomplexSpectralSequence
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: E = BicomplexSpectralSequence(l)
sage: E.group(2,0,1)
Additive abelian group isomorphic to Z/2 + Z
sage: E.table(3,0,2,0,2)
0           0   0
Z/2 + Z/4   0   0
0           0   Z
sage: E.matrix(2,2,0)
[ 0  0]
[-4  0]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import BicomplexSpectralSequence
>>> C1 = ChainComplex({Integer(1): matrix(ZZ, Integer(0), Integer(2), [])},␣
↪degree_of_differential=-Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> C2 = ChainComplex({Integer(1): matrix(ZZ, Integer(1), Integer(2), [Integer(1),
↪ Integer(0)])},degree_of_differential=-Integer(1))
>>> C3 = ChainComplex({Integer(0): matrix(ZZ, Integer(0),Integer(2) , [])},degree_
↪of_differential=-Integer(1))
>>> M1 = Hom(C2,C1)({Integer(1): matrix(ZZ, Integer(2), Integer(2), [Integer(2),␣
↪Integer(0), Integer(0), Integer(2)])})
>>> M2 = Hom(C3,C2)({Integer(0): matrix(ZZ, Integer(1), Integer(2), [Integer(2),␣
↪Integer(0)])})
>>> l = [M1, M2]
>>> E = BicomplexSpectralSequence(l)
>>> E.group(Integer(2),Integer(0),Integer(1))
Additive abelian group isomorphic to Z/2 + Z
>>> E.table(Integer(3),Integer(0),Integer(2),Integer(0),Integer(2))
0           0   0
Z/2 + Z/4   0   0
0           0   Z
>>> E.matrix(Integer(2),Integer(2),Integer(0))
[ 0  0]
[-4  0]
```

sage.interfaces.kenzo.**EilenbergMacLaneSpace**(G, n)

Return the Eilenberg-MacLane space `K(G, n)` as a Kenzo simplicial group.

The Eilenberg-MacLane space `K(G, n)` is the space whose has n'th homotopy group isomorphic to `G`, and the rest of the homotopy groups are trivial.

INPUT:

- `G` – group; currently only `ZZ` and the additive group of two elements are supported
- `n` – the dimension in which the homotopy is not trivial

OUTPUT: a *KenzoSimplicialGroup*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import EilenbergMacLaneSpace
sage: e3 = EilenbergMacLaneSpace(ZZ, 3)
sage: [e3.homology(i) for i in range(8)]
[Z, 0, 0, Z, 0, C2, 0, C3]
sage: f3 = EilenbergMacLaneSpace(AdditiveAbelianGroup([2]), 3)
sage: [f3.homology(i) for i in range(8)]
[Z, 0, 0, C2, 0, C2, C2, C2]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import EilenbergMacLaneSpace
>>> e3 = EilenbergMacLaneSpace(ZZ, Integer(3))
>>> [e3.homology(i) for i in range(Integer(8))]
[Z, 0, 0, Z, 0, C2, 0, C3]
>>> f3 = EilenbergMacLaneSpace(AdditiveAbelianGroup([Integer(2)]), Integer(3))
>>> [f3.homology(i) for i in range(Integer(8))]
[Z, 0, 0, C2, 0, C2, C2, C2]
```

sage.interfaces.kenzo.**KAbstractSimplex**(*simplex*)

Convert an AbstractSimplex in Sage to an abstract simplex of Kenzo.

INPUT:

- `simplex` – an AbstractSimplex

OUTPUT: an abstract simplex of Kenzo

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.topology.simplicial_set import AbstractSimplex
sage: from sage.interfaces.kenzo import (
....:   KAbstractSimplex, SAbstractSimplex)
sage: SAbSm = AbstractSimplex(1, (2,0,3,2,1), name = 'SAbSm')
sage: KAbSm = KAbstractSimplex(SAbSm)
sage: SAbSm2 = SAbstractSimplex(KAbSm, 1)
sage: SAbSm.degeneracies() == SAbSm2.degeneracies()
True
sage: SAbSm.dimension() == SAbSm2.dimension()
True
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.topology.simplicial_set import AbstractSimplex
>>> from sage.interfaces.kenzo import (
...   KAbstractSimplex, SAbstractSimplex)
>>> SAbSm = AbstractSimplex(Integer(1), (Integer(2),Integer(0),Integer(3),
↪Integer(2),Integer(1)), name = 'SAbSm')
>>> KAbSm = KAbstractSimplex(SAbSm)
>>> SAbSm2 = SAbstractSimplex(KAbSm, Integer(1))
>>> SAbSm.degeneracies() == SAbSm2.degeneracies()
True
>>> SAbSm.dimension() == SAbSm2.dimension()
True
```

sage.interfaces.kenzo.**KChainComplex**(*chain_complex*)

Construct a KenzoChainComplex from a ChainComplex of degree -1 in Sage.

INPUT:

- `chain_complex` – a ChainComplex of degree -1

OUTPUT: a KenzoChainComplex

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import KChainComplex
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)
sage: kenzo_chcm = KChainComplex(sage_chcm)
sage: kenzo_chcm
[K... Chain-Complex]
```

(continues on next page)

(continued from previous page)

```
sage: kenzo_chcm.homology(5)
Z x Z
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import KChainComplex
>>> m1 = matrix(ZZ, Integer(3), Integer(2), [-Integer(1), Integer(1), Integer(3),␣
↪-Integer(4), Integer(5), Integer(6)])
>>> m4 = matrix(ZZ, Integer(2), Integer(2), [Integer(1), Integer(2), Integer(3),␣
↪Integer(6)])
>>> m5 = matrix(ZZ, Integer(2), Integer(3), [Integer(2), Integer(2), Integer(2), -
↪Integer(1), -Integer(1), -Integer(1)])
>>> sage_chcm = ChainComplex({Integer(1): m1, Integer(4): m4, Integer(5): m5},␣
↪degree = -Integer(1))
>>> kenzo_chcm = KChainComplex(sage_chcm)
>>> kenzo_chcm
[K... Chain-Complex]
>>> kenzo_chcm.homology(Integer(5))
Z x Z
```

`sage.interfaces.kenzo.KChainComplexMorphism(morphism)`

Construct a KenzoChainComplexMorphism from a ChainComplexMorphism in Sage.

INPUT:

- `morphism` – a morphism of chain complexes

OUTPUT: a *KenzoChainComplexMorphism*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import KChainComplexMorphism
sage: C = ChainComplex({0: identity_matrix(ZZ, 1)})
sage: D = ChainComplex({0: zero_matrix(ZZ, 1), 1: zero_matrix(ZZ, 1)})
sage: f = Hom(C,D)({0: identity_matrix(ZZ, 1), 1: zero_matrix(ZZ, 1)})
sage: g = KChainComplexMorphism(f); g
[K... Morphism (degree 0): K... -> K...]
sage: g.source_complex()
[K... Chain-Complex]
sage: g.target_complex()
[K... Chain-Complex]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import KChainComplexMorphism
>>> C = ChainComplex({Integer(0): identity_matrix(ZZ, Integer(1))})
>>> D = ChainComplex({Integer(0): zero_matrix(ZZ, Integer(1)), Integer(1): zero_
↪matrix(ZZ, Integer(1))})
>>> f = Hom(C,D)({Integer(0): identity_matrix(ZZ, Integer(1)), Integer(1): zero_
↪matrix(ZZ, Integer(1))})
>>> g = KChainComplexMorphism(f); g
[K... Morphism (degree 0): K... -> K...]
```

(continues on next page)

(continued from previous page)

```
>>> g.source_complex()
[K... Chain-Complex]
>>> g.target_complex()
[K... Chain-Complex]
```

sage.interfaces.kenzo.**KFiniteSimplicialSet**(*sset*)

Convert a finite SimplicialSet in Sage to a finite simplicial set of Kenzo.

INPUT:

- `sset` – a finite SimplicialSet

OUTPUT: a finite simplicial set of Kenzo

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: from sage.interfaces.kenzo import KFiniteSimplicialSet
sage: s0 = AbstractSimplex(0, name='s0')
sage: s1 = AbstractSimplex(0, name='s1')
sage: s2 = AbstractSimplex(0, name='s2')
sage: s01 = AbstractSimplex(1, name='s01')
sage: s02 = AbstractSimplex(1, name='s02')
sage: s12 = AbstractSimplex(1, name='s12')
sage: s012 = AbstractSimplex(2, name='s012')
sage: Triangle = SimplicialSet({s01: (s1, s0),\
....: s02: (s2, s0), s12: (s2, s1)}, base_point = s0)
sage: KTriangle = KFiniteSimplicialSet(Triangle)
sage: KTriangle.homology(1)
Z
sage: KTriangle.basis(1)
['CELL_1_0', 'CELL_1_1', 'CELL_1_2']
sage: S1 = simplicial_sets.Sphere(1)
sage: S3 = simplicial_sets.Sphere(3)
sage: KS1vS3 = KFiniteSimplicialSet(S1.wedge(S3))
sage: KS1vS3.homology(3)
Z
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
>>> from sage.interfaces.kenzo import KFiniteSimplicialSet
>>> s0 = AbstractSimplex(Integer(0), name='s0')
>>> s1 = AbstractSimplex(Integer(0), name='s1')
>>> s2 = AbstractSimplex(Integer(0), name='s2')
>>> s01 = AbstractSimplex(Integer(1), name='s01')
>>> s02 = AbstractSimplex(Integer(1), name='s02')
>>> s12 = AbstractSimplex(Integer(1), name='s12')
>>> s012 = AbstractSimplex(Integer(2), name='s012')
>>> Triangle = SimplicialSet({s01: (s1, s0),s02: (s2, s0), s12: (s2, s1)}, base_
↪point = s0)
>>> KTriangle = KFiniteSimplicialSet(Triangle)
>>> KTriangle.homology(Integer(1))
```

(continues on next page)

(continued from previous page)

```
Z
>>> KTriangle.basis(Integer(1))
['CELL_1_0', 'CELL_1_1', 'CELL_1_2']
>>> S1 = simplicial_sets.Sphere(Integer(1))
>>> S3 = simplicial_sets.Sphere(Integer(3))
>>> KS1vS3 = KFiniteSimplicialSet(S1.wedge(S3))
>>> KS1vS3.homology(Integer(3))
Z
```

class sage.interfaces.kenzo.**KenzoChainComplex**(*kenzo_object*)

Bases: *KenzoObject*

Wrapper to Kenzo chain complexes. Kenzo simplicial sets are a particular case of Kenzo chain complexes.

basis(*dim*)

Return the list of generators of the chain complex associated to the kenzo object `self` in dimension `dim`.

INPUT:

- `dim` – integer

OUTPUT: list of the form ['G"dim"G0', 'G"dim"G1', 'G"dim"G2', …]

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import KChainComplex
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)
sage: kenzo_chcm = KChainComplex(sage_chcm)
sage: kenzo_chcm
[K... Chain-Complex]
sage: for i in range(6):
....:     print("Basis in dimension %i: %s" % (i, kenzo_chcm.basis(i)))
Basis in dimension 0: ['G0G0', 'G0G1', 'G0G2']
Basis in dimension 1: ['G1G0', 'G1G1']
Basis in dimension 2: None
Basis in dimension 3: ['G3G0', 'G3G1']
Basis in dimension 4: ['G4G0', 'G4G1']
Basis in dimension 5: ['G5G0', 'G5G1', 'G5G2']
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import KChainComplex
>>> m1 = matrix(ZZ, Integer(3), Integer(2), [-Integer(1), Integer(1),␣
↪Integer(3), -Integer(4), Integer(5), Integer(6)])
>>> m4 = matrix(ZZ, Integer(2), Integer(2), [Integer(1), Integer(2),␣
↪Integer(3), Integer(6)])
>>> m5 = matrix(ZZ, Integer(2), Integer(3), [Integer(2), Integer(2),␣
↪Integer(2), -Integer(1), -Integer(1), -Integer(1)])
>>> sage_chcm = ChainComplex({Integer(1): m1, Integer(4): m4, Integer(5): m5},
↪ degree = -Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> kenzo_chcm = KChainComplex(sage_chcm)
>>> kenzo_chcm
[K... Chain-Complex]
>>> for i in range(Integer(6)):
...     print("Basis in dimension %i: %s" % (i, kenzo_chcm.basis(i)))
Basis in dimension 0: ['G0G0', 'G0G1', 'G0G2']
Basis in dimension 1: ['G1G0', 'G1G1']
Basis in dimension 2: None
Basis in dimension 3: ['G3G0', 'G3G1']
Basis in dimension 4: ['G4G0', 'G4G1']
Basis in dimension 5: ['G5G0', 'G5G1', 'G5G2']
```

differential(*dim=None*, *comb=None*)

Return the differential of a combination.

INPUT:

- `dim` – integer or `None` (default)
- `comb` – list representing a formal sum of generators in the module of dimension `dim` or `None` (default). For example, to represent G7G12 + 3*G7G0 - 5*G7G3 we use the list [3, 'G7G0', -5, 'G7G3', 1, 'G7G12']. Note that the generators must be in ascending order respect to the number after the second G in their representation; the parameter `comb` = [1, 'G7G12', 3, 'G7G0', -5, 'G7G3'] will produce an error in Kenzo.

OUTPUT:

- If `dim` and `comb` are not `None`, it returns a Kenzo combination representing the differential of the formal combination represented by `comb` in the chain complex `self` in dimension `dim`. On the other hand, if `dim` or `comb` (or both) take `None` value, the differential `KenzoMorphismChainComplex` of `self` is returned.

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import KChainComplex
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)
sage: kenzo_chcm = KChainComplex(sage_chcm)
sage: kenzo_chcm
[K... Chain-Complex]
sage: kenzo_chcm.basis(4)
['G4G0', 'G4G1']
sage: kenzo_chcm.differential(4, [1, 'G4G0'])

----------------------------------------------------------------------{CMBN 3}
<1 * G3G0>
<3 * G3G1>
------------------------------------------------------------------------------

sage: kenzo_chcm.basis(5)
['G5G0', 'G5G1', 'G5G2']
sage: kenzo_chcm.differential(5, [1, 'G5G0', 2, 'G5G2'])
```

(continues on next page)

(continued from previous page)

```
-----------------------------------------------------------------------{CMBN 4}
<6 * G4G0>
<-3 * G4G1>
------------------------------------------------------------------------------
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import KChainComplex
>>> m1 = matrix(ZZ, Integer(3), Integer(2), [-Integer(1), Integer(1),␣
↪Integer(3), -Integer(4), Integer(5), Integer(6)])
>>> m4 = matrix(ZZ, Integer(2), Integer(2), [Integer(1), Integer(2),␣
↪Integer(3), Integer(6)])
>>> m5 = matrix(ZZ, Integer(2), Integer(3), [Integer(2), Integer(2),␣
↪Integer(2), -Integer(1), -Integer(1), -Integer(1)])
>>> sage_chcm = ChainComplex({Integer(1): m1, Integer(4): m4, Integer(5): m5},
↪ degree = -Integer(1))
>>> kenzo_chcm = KChainComplex(sage_chcm)
>>> kenzo_chcm
[K... Chain-Complex]
>>> kenzo_chcm.basis(Integer(4))
['G4G0', 'G4G1']
>>> kenzo_chcm.differential(Integer(4), [Integer(1), 'G4G0'])
<BLANKLINE>
-----------------------------------------------------------------------{CMBN 3}
<1 * G3G0>
<3 * G3G1>
------------------------------------------------------------------------------
<BLANKLINE>
>>> kenzo_chcm.basis(Integer(5))
['G5G0', 'G5G1', 'G5G2']
>>> kenzo_chcm.differential(Integer(5), [Integer(1), 'G5G0', Integer(2), 'G5G2
↪'])
<BLANKLINE>
-----------------------------------------------------------------------{CMBN 4}
<6 * G4G0>
<-3 * G4G1>
------------------------------------------------------------------------------
<BLANKLINE>
```

homology(*n*)

Return the n-th homology group of the chain complex associated to this kenzo object.

INPUT:

- n – the dimension in which compute the homology

OUTPUT: a homology group

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: s2 = Sphere(2)
```

(continues on next page)

(continued from previous page)

```
sage: s2
[K1 Simplicial-Set]
sage: s2.homology(2)
Z
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> s2 = Sphere(Integer(2))
>>> s2
[K1 Simplicial-Set]
>>> s2.homology(Integer(2))
Z
```

identity_morphism()

Return the identity morphism (degree 0) between `self` and itself.

OUTPUT: a *KenzoChainComplexMorphism*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: s2 = Sphere(2)
sage: tp = s2.tensor_product(s2)
sage: idnt = tp.identity_morphism()
sage: type(idnt)
<class 'sage.interfaces.kenzo.KenzoChainComplexMorphism'>
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> s2 = Sphere(Integer(2))
>>> tp = s2.tensor_product(s2)
>>> idnt = tp.identity_morphism()
>>> type(idnt)
<class 'sage.interfaces.kenzo.KenzoChainComplexMorphism'>
```

null_morphism(*target=None*, *degree=None*)

Return the null morphism between the chain complexes `self` and `target` of degree `degree`.

INPUT:

- `target` – a KenzoChainComplex or `None` (default)
- `degree` – integer or `None` (default)

OUTPUT:

- A *KenzoChainComplexMorphism* representing the null morphism between `self` and `target` of degree `degree`. If `target` takes `None` value, `self` is assumed as the target chain complex; if `degree` takes `None` value, 0 is assumed as the degree of the null morphism.

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: s2 = Sphere(2)
sage: s3 = Sphere(3)
sage: tp22 = s2.tensor_product(s2)
sage: tp22
[K... Chain-Complex]
sage: tp23 = s2.tensor_product(s3)
sage: tp23
[K... Chain-Complex]
sage: null1 = tp22.null_morphism()
sage: null1
[K... Morphism (degree 0): K... -> K...]
sage: null2 = tp22.null_morphism(target = tp23, degree = -3)
sage: null2
[K... Morphism (degree -3): K... -> K...]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> s2 = Sphere(Integer(2))
>>> s3 = Sphere(Integer(3))
>>> tp22 = s2.tensor_product(s2)
>>> tp22
[K... Chain-Complex]
>>> tp23 = s2.tensor_product(s3)
>>> tp23
[K... Chain-Complex]
>>> null1 = tp22.null_morphism()
>>> null1
[K... Morphism (degree 0): K... -> K...]
>>> null2 = tp22.null_morphism(target = tp23, degree = -Integer(3))
>>> null2
[K... Morphism (degree -3): K... -> K...]
```

orgn()

Return the :orgn slot of Kenzo, which stores as a list the origin of the object.

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: s2 = Sphere(2)
sage: l2 = s2.loop_space()
sage: l2.orgn()
'(LOOP-SPACE [K... Simplicial-Set])'
sage: A = l2.cartesian_product(s2)
sage: A.orgn()
'(CRTS-PRDC [K... Simplicial-Group] [K... Simplicial-Set])'
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
```

(continues on next page)

(continued from previous page)

```
>>> s2 = Sphere(Integer(2))
>>> l2 = s2.loop_space()
>>> l2.orgn()
'(LOOP-SPACE [K... Simplicial-Set])'
>>> A = l2.cartesian_product(s2)
>>> A.orgn()
'(CRTS-PRDC [K... Simplicial-Group] [K... Simplicial-Set])'
```

tensor_product(*other*)

Return the tensor product of `self` and `other`.

INPUT:

- `other` – The Kenzo object with which to compute the tensor product

OUTPUT: a *KenzoChainComplex*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: s2 = Sphere(2)
sage: s3 = Sphere(3)
sage: p = s2.tensor_product(s3)
sage: type(p)
<class 'sage.interfaces.kenzo.KenzoChainComplex'>
sage: [p.homology(i) for i in range(8)]
[Z, 0, Z, Z, 0, Z, 0, 0]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> s2 = Sphere(Integer(2))
>>> s3 = Sphere(Integer(3))
>>> p = s2.tensor_product(s3)
>>> type(p)
<class 'sage.interfaces.kenzo.KenzoChainComplex'>
>>> [p.homology(i) for i in range(Integer(8))]
[Z, 0, Z, Z, 0, Z, 0, 0]
```

class sage.interfaces.kenzo.**KenzoChainComplexMorphism**(*kenzo_object*)

Bases: *KenzoObject*

Wrapper to Kenzo morphisms between chain complexes.

change_source_target_complex(*source=None*, *target=None*)

Build, from the morphism `self`, a new morphism with `source` and `target` as source and target Kenzo chain complexes, respectively.

INPUT:

- `source` – a KenzoChainComplex instance or `None` (default)
- `target` – a KenzoChainComplex instance or `None` (default)

OUTPUT:

- A *KenzoChainComplexMorphism* inheriting from `self` the degree (:degr slot in Kenzo), the algorithm (:intr slot in Kenzo) and the strategy (:strt slot in Kenzo). The source and target slots of this new morphism are given by the parameters `source` and `target` respectively; if any parameter is ommited, the corresponding slot is inherited from `self`.

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere, KenzoChainComplex
sage: from sage.libs.ecl import ecl_eval
sage: ZCC = KenzoChainComplex(ecl_eval("(z-chcm)"))
sage: ZCC
[K... Chain-Complex]
sage: s2 = Sphere(2)
sage: s3 = Sphere(3)
sage: tp = s2.tensor_product(s3)
sage: tp
[K... Filtered-Chain-Complex]
sage: null = ZCC.null_morphism(tp)
sage: null
[K... Morphism (degree 0): K... -> K...]
sage: null.source_complex()
[K... Chain-Complex]
sage: null2 = null.change_source_target_complex(source = tp)
sage: null2
[K... Morphism (degree 0): K... -> K...]
sage: null2.source_complex()
[K... Filtered-Chain-Complex]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere, KenzoChainComplex
>>> from sage.libs.ecl import ecl_eval
>>> ZCC = KenzoChainComplex(ecl_eval("(z-chcm)"))
>>> ZCC
[K... Chain-Complex]
>>> s2 = Sphere(Integer(2))
>>> s3 = Sphere(Integer(3))
>>> tp = s2.tensor_product(s3)
>>> tp
[K... Filtered-Chain-Complex]
>>> null = ZCC.null_morphism(tp)
>>> null
[K... Morphism (degree 0): K... -> K...]
>>> null.source_complex()
[K... Chain-Complex]
>>> null2 = null.change_source_target_complex(source = tp)
>>> null2
[K... Morphism (degree 0): K... -> K...]
>>> null2.source_complex()
[K... Filtered-Chain-Complex]
```

composite(*object=None*)

Return the composite of `self` and the morphism(s) given by the parameter `object`.

INPUT:

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

OUTPUT:

- A *KenzoChainComplexMorphism*: if `object` is a KenzoChainComplexMorphism, the composite of `self` and `object` is returned; if `object` is a KenzoChainComplex, the composite of `self` and the differential morphism of `object` is returned; if `object` is a tuple, the composite of `self` and the morphisms or the differential morphisms of the given chain complexes in `object` is returned (if `object` is `None`, `self` morphism is returned).

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: s2 = Sphere(2)
sage: s3 = Sphere(3)
sage: tp22 = s2.tensor_product(s2)
sage: tp23 = s2.tensor_product(s3)
sage: idnt = tp22.identity_morphism()
sage: idnt
[K... Morphism (degree 0): K... -> K...]
sage: null = tp23.null_morphism(target = tp22, degree = 4)
sage: null
[K... Morphism (degree 4): K... -> K...]
sage: idnt.composite((tp22, null))
[K... Morphism (degree 3): K... -> K...]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> s2 = Sphere(Integer(2))
>>> s3 = Sphere(Integer(3))
>>> tp22 = s2.tensor_product(s2)
>>> tp23 = s2.tensor_product(s3)
>>> idnt = tp22.identity_morphism()
>>> idnt
[K... Morphism (degree 0): K... -> K...]
>>> null = tp23.null_morphism(target = tp22, degree = Integer(4))
>>> null
[K... Morphism (degree 4): K... -> K...]
>>> idnt.composite((tp22, null))
[K... Morphism (degree 3): K... -> K...]
```

degree()

Return the degree of the morphism.

OUTPUT: integer; the degree of the morphism

EXAMPLES:

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

(continues on next page)

(continued from previous page)

```
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)
sage: kenzo_chcm = KChainComplex(sage_chcm)
sage: kenzo_chcm
[K... Chain-Complex]
sage: differential_morphism = kenzo_chcm.differential()
sage: differential_morphism
[K... Morphism (degree -1): K... -> K...]
sage: differential_morphism.degree()
-1
sage: differential_morphism.composite(differential_morphism).degree()
-2
sage: kenzo_chcm.null_morphism().degree()
0
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import KChainComplex
>>> m1 = matrix(ZZ, Integer(3), Integer(2), [-Integer(1), Integer(1),␣
↪Integer(3), -Integer(4), Integer(5), Integer(6)])
>>> m4 = matrix(ZZ, Integer(2), Integer(2), [Integer(1), Integer(2),␣
↪Integer(3), Integer(6)])
>>> m5 = matrix(ZZ, Integer(2), Integer(3), [Integer(2), Integer(2),␣
↪Integer(2), -Integer(1), -Integer(1), -Integer(1)])
>>> sage_chcm = ChainComplex({Integer(1): m1, Integer(4): m4, Integer(5): m5},
↪ degree=-Integer(1))
>>> kenzo_chcm = KChainComplex(sage_chcm)
>>> kenzo_chcm
[K... Chain-Complex]
>>> differential_morphism = kenzo_chcm.differential()
>>> differential_morphism
[K... Morphism (degree -1): K... -> K...]
>>> differential_morphism.degree()
-1
>>> differential_morphism.composite(differential_morphism).degree()
-2
>>> kenzo_chcm.null_morphism().degree()
0
```

destructive_change_source_target_complex(*source=None*, *target=None*)

Modify destructively the morphism `self` taking `source` and `target` as source and target Kenzo chain complexes of `self`, respectively.

INPUT:

- `source` – a KenzoChainComplex instance or `None` (default)
- `target` – a KenzoChainComplex instance or `None` (default)

OUTPUT:

- A *KenzoChainComplexMorphism*. The source and target slots of `self` are replaced respectively by the parameters `source` and `target`; if any parameter is ommited, the corresponding slot is inherited from `self`.

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere, KenzoChainComplex
sage: from sage.libs.ecl import ecl_eval
sage: ZCC = KenzoChainComplex(ecl_eval("(z-chcm)"))
sage: ZCC
[K... Chain-Complex]
sage: s2 = Sphere(2)
sage: s3 = Sphere(3)
sage: tp = s2.tensor_product(s3)
sage: tp
[K... Filtered-Chain-Complex]
sage: null = ZCC.null_morphism(tp)
sage: null
[K... Morphism (degree 0): K... -> K...]
sage: null.target_complex()
[K... Filtered-Chain-Complex]
sage: null.destructive_change_source_target_complex(target = ZCC)
[K... Cohomology-Class on K... of degree 0]
sage: null.target_complex()
[K... Chain-Complex]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere, KenzoChainComplex
>>> from sage.libs.ecl import ecl_eval
>>> ZCC = KenzoChainComplex(ecl_eval("(z-chcm)"))
>>> ZCC
[K... Chain-Complex]
>>> s2 = Sphere(Integer(2))
>>> s3 = Sphere(Integer(3))
>>> tp = s2.tensor_product(s3)
>>> tp
[K... Filtered-Chain-Complex]
>>> null = ZCC.null_morphism(tp)
>>> null
[K... Morphism (degree 0): K... -> K...]
>>> null.target_complex()
[K... Filtered-Chain-Complex]
>>> null.destructive_change_source_target_complex(target = ZCC)
[K... Cohomology-Class on K... of degree 0]
>>> null.target_complex()
[K... Chain-Complex]
```

evaluation(*dim*, *comb*)

Apply the morphism on a combination `comb` of dimension `dim`.

INPUT:

- `dim` – integer
- `comb` – list representing a formal sum of generators in the module of dimension `dim`. For example, to represent G7G12 + 3*G7G0 - 5*G7G3 we use the list [3, 'G7G0', -5, 'G7G3', 1, 'G7G12']. Note that the generators must be in ascending order respect to the number after the second G in their representation;

the parameter `comb` = [1, 'G7G12', 3, 'G7G0', -5, 'G7G3'] will produce an error in Kenzo.

OUTPUT:

- A Kenzo combination representing the result of applying the morphism on the formal combination represented by `comb` in the chain complex `self` in dimension `dim`.

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import KChainComplex
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)
sage: kenzo_chcm = KChainComplex(sage_chcm)
sage: kenzo_chcm
[K... Chain-Complex]
sage: differential_morphism = kenzo_chcm.differential()
sage: differential_morphism
[K... Morphism (degree -1): K... -> K...]
sage: dif_squared = differential_morphism.composite(differential_morphism)
sage: dif_squared
[K... Morphism (degree -2): K... -> K...]
sage: kenzo_chcm.basis(5)
['G5G0', 'G5G1', 'G5G2']
sage: kenzo_chcm.differential(5, [1, 'G5G0', 2, 'G5G2'])

----------------------------------------------------------------------{CMBN 4}
<6 * G4G0>
<-3 * G4G1>
------------------------------------------------------------------------------

sage: differential_morphism.evaluation(5, [1, 'G5G0', 2, 'G5G2'])

----------------------------------------------------------------------{CMBN 4}
<6 * G4G0>
<-3 * G4G1>
------------------------------------------------------------------------------

sage: dif_squared.evaluation(5, [1, 'G5G0', 2, 'G5G2'])

----------------------------------------------------------------------{CMBN 3}
------------------------------------------------------------------------------

sage: idnt = kenzo_chcm.identity_morphism()
sage: idx2 = idnt.sum(idnt)
sage: idnt.evaluation(5, [1, 'G5G0', 2, 'G5G2'])

----------------------------------------------------------------------{CMBN 5}
<1 * G5G0>
<2 * G5G2>
------------------------------------------------------------------------------

sage: idx2.evaluation(5, [1, 'G5G0', 2, 'G5G2'])
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------{CMBN 5}
<2 * G5G0>
<4 * G5G2>
------------------------------------------------------------------------------
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import KChainComplex
>>> m1 = matrix(ZZ, Integer(3), Integer(2), [-Integer(1), Integer(1),␣
↪Integer(3), -Integer(4), Integer(5), Integer(6)])
>>> m4 = matrix(ZZ, Integer(2), Integer(2), [Integer(1), Integer(2),␣
↪Integer(3), Integer(6)])
>>> m5 = matrix(ZZ, Integer(2), Integer(3), [Integer(2), Integer(2),␣
↪Integer(2), -Integer(1), -Integer(1), -Integer(1)])
>>> sage_chcm = ChainComplex({Integer(1): m1, Integer(4): m4, Integer(5): m5},
↪ degree = -Integer(1))
>>> kenzo_chcm = KChainComplex(sage_chcm)
>>> kenzo_chcm
[K... Chain-Complex]
>>> differential_morphism = kenzo_chcm.differential()
>>> differential_morphism
[K... Morphism (degree -1): K... -> K...]
>>> dif_squared = differential_morphism.composite(differential_morphism)
>>> dif_squared
[K... Morphism (degree -2): K... -> K...]
>>> kenzo_chcm.basis(Integer(5))
['G5G0', 'G5G1', 'G5G2']
>>> kenzo_chcm.differential(Integer(5), [Integer(1), 'G5G0', Integer(2), 'G5G2
↪'])
<BLANKLINE>
----------------------------------------------------------------------{CMBN 4}
<6 * G4G0>
<-3 * G4G1>
------------------------------------------------------------------------------
<BLANKLINE>
>>> differential_morphism.evaluation(Integer(5), [Integer(1), 'G5G0',␣
↪Integer(2), 'G5G2'])
<BLANKLINE>
----------------------------------------------------------------------{CMBN 4}
<6 * G4G0>
<-3 * G4G1>
------------------------------------------------------------------------------
<BLANKLINE>
>>> dif_squared.evaluation(Integer(5), [Integer(1), 'G5G0', Integer(2), 'G5G2
↪'])
<BLANKLINE>
----------------------------------------------------------------------{CMBN 3}
------------------------------------------------------------------------------
<BLANKLINE>
>>> idnt = kenzo_chcm.identity_morphism()
>>> idx2 = idnt.sum(idnt)
```

(continues on next page)

(continued from previous page)

```
>>> idnt.evaluation(Integer(5), [Integer(1), 'G5G0', Integer(2), 'G5G2'])
<BLANKLINE>
----------------------------------------------------------------------{CMBN 5}
<1 * G5G0>
<2 * G5G2>
------------------------------------------------------------------------------
<BLANKLINE>
>>> idx2.evaluation(Integer(5), [Integer(1), 'G5G0', Integer(2), 'G5G2'])
<BLANKLINE>
----------------------------------------------------------------------{CMBN 5}
<2 * G5G0>
<4 * G5G2>
------------------------------------------------------------------------------
<BLANKLINE>
```

opposite()

Return the opposite morphism of `self`, i.e., -1 x `self`.

OUTPUT: a *KenzoChainComplexMorphism*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import KChainComplex
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)
sage: kenzo_chcm = KChainComplex(sage_chcm)
sage: kenzo_chcm
[K... Chain-Complex]
sage: idnt = kenzo_chcm.identity_morphism()
sage: idnt
[K... Morphism (degree 0): K... -> K...]
sage: opps_id = idnt.opposite()
sage: opps_id
[K... Morphism (degree 0): K... -> K...]
sage: kenzo_chcm.basis(4)
['G4G0', 'G4G1']
sage: idnt.evaluation(4, [2, 'G4G0', -5, 'G4G1'])

----------------------------------------------------------------------{CMBN 4}
<2 * G4G0>
<-5 * G4G1>
------------------------------------------------------------------------------

sage: opps_id.evaluation(4, [2, 'G4G0', -5, 'G4G1'])

----------------------------------------------------------------------{CMBN 4}
<-2 * G4G0>
<5 * G4G1>
------------------------------------------------------------------------------
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import KChainComplex
>>> m1 = matrix(ZZ, Integer(3), Integer(2), [-Integer(1), Integer(1),␣
↪Integer(3), -Integer(4), Integer(5), Integer(6)])
>>> m4 = matrix(ZZ, Integer(2), Integer(2), [Integer(1), Integer(2),␣
↪Integer(3), Integer(6)])
>>> m5 = matrix(ZZ, Integer(2), Integer(3), [Integer(2), Integer(2),␣
↪Integer(2), -Integer(1), -Integer(1), -Integer(1)])
>>> sage_chcm = ChainComplex({Integer(1): m1, Integer(4): m4, Integer(5): m5},
↪ degree = -Integer(1))
>>> kenzo_chcm = KChainComplex(sage_chcm)
>>> kenzo_chcm
[K... Chain-Complex]
>>> idnt = kenzo_chcm.identity_morphism()
>>> idnt
[K... Morphism (degree 0): K... -> K...]
>>> opps_id = idnt.opposite()
>>> opps_id
[K... Morphism (degree 0): K... -> K...]
>>> kenzo_chcm.basis(Integer(4))
['G4G0', 'G4G1']
>>> idnt.evaluation(Integer(4), [Integer(2), 'G4G0', -Integer(5), 'G4G1'])
<BLANKLINE>
----------------------------------------------------------------------{CMBN 4}
<2 * G4G0>
<-5 * G4G1>
------------------------------------------------------------------------------
<BLANKLINE>
>>> opps_id.evaluation(Integer(4), [Integer(2), 'G4G0', -Integer(5), 'G4G1'])
<BLANKLINE>
----------------------------------------------------------------------{CMBN 4}
<-2 * G4G0>
<5 * G4G1>
------------------------------------------------------------------------------
<BLANKLINE>
```

source_complex()

Return the source chain complex of the morphism.

OUTPUT: a *KenzoChainComplex*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import KChainComplex
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)
sage: kenzo_chcm = KChainComplex(sage_chcm)
sage: kenzo_chcm
[K... Chain-Complex]
sage: differential_morphism = kenzo_chcm.differential()
```

(continues on next page)

(continued from previous page)

```
sage: differential_morphism
[K... Morphism (degree -1): K... -> K...]
sage: differential_morphism.source_complex()
[K... Chain-Complex]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import KChainComplex
>>> m1 = matrix(ZZ, Integer(3), Integer(2), [-Integer(1), Integer(1),␣
↪Integer(3), -Integer(4), Integer(5), Integer(6)])
>>> m4 = matrix(ZZ, Integer(2), Integer(2), [Integer(1), Integer(2),␣
↪Integer(3), Integer(6)])
>>> m5 = matrix(ZZ, Integer(2), Integer(3), [Integer(2), Integer(2),␣
↪Integer(2), -Integer(1), -Integer(1), -Integer(1)])
>>> sage_chcm = ChainComplex({Integer(1): m1, Integer(4): m4, Integer(5): m5},
↪ degree = -Integer(1))
>>> kenzo_chcm = KChainComplex(sage_chcm)
>>> kenzo_chcm
[K... Chain-Complex]
>>> differential_morphism = kenzo_chcm.differential()
>>> differential_morphism
[K... Morphism (degree -1): K... -> K...]
>>> differential_morphism.source_complex()
[K... Chain-Complex]
```

substract (*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: # optional - kenzo
sage: from sage.interfaces.kenzo import KChainComplex
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)
sage: kenzo_chcm = KChainComplex(sage_chcm)
sage: kenzo_chcm
[K... Chain-Complex]
sage: idnt = kenzo_chcm.identity_morphism()
sage: idnt
[K... Morphism (degree 0): K... -> K...]
sage: opps_id = idnt.opposite()
```

(continues on next page)

(continued from previous page)

```
sage: opps_id
[K... Morphism (degree 0): K... -> K...]
sage: null = kenzo_chcm.null_morphism()
sage: null
[K... Morphism (degree 0): K... -> K...]
sage: idx2 = idnt.substract(opps_id)
sage: opps_idx2 = idx2.substract(
....:   (opps_id, idnt, idnt, null, idx2.substract(opps_id)))
sage: kenzo_chcm.basis(4)
['G4G0', 'G4G1']
sage: idx2.evaluation(4, [2, 'G4G0', -5, 'G4G1'])

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

sage: opps_idx2.evaluation(4, [2, 'G4G0', -5, 'G4G1'])

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

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import KChainComplex
>>> m1 = matrix(ZZ, Integer(3), Integer(2), [-Integer(1), Integer(1),␣
↪Integer(3), -Integer(4), Integer(5), Integer(6)])
>>> m4 = matrix(ZZ, Integer(2), Integer(2), [Integer(1), Integer(2),␣
↪Integer(3), Integer(6)])
>>> m5 = matrix(ZZ, Integer(2), Integer(3), [Integer(2), Integer(2),␣
↪Integer(2), -Integer(1), -Integer(1), -Integer(1)])
>>> sage_chcm = ChainComplex({Integer(1): m1, Integer(4): m4, Integer(5): m5},
↪ degree = -Integer(1))
>>> kenzo_chcm = KChainComplex(sage_chcm)
>>> kenzo_chcm
[K... Chain-Complex]
>>> idnt = kenzo_chcm.identity_morphism()
>>> idnt
[K... Morphism (degree 0): K... -> K...]
>>> opps_id = idnt.opposite()
>>> opps_id
[K... Morphism (degree 0): K... -> K...]
>>> null = kenzo_chcm.null_morphism()
>>> null
[K... Morphism (degree 0): K... -> K...]
>>> idx2 = idnt.substract(opps_id)
>>> opps_idx2 = idx2.substract(
...   (opps_id, idnt, idnt, null, idx2.substract(opps_id)))
>>> kenzo_chcm.basis(Integer(4))
['G4G0', 'G4G1']
```

(continues on next page)

(continued from previous page)

```
>>> idx2.evaluation(Integer(4), [Integer(2), 'G4G0', -Integer(5), 'G4G1'])
<BLANKLINE>
----------------------------------------------------------------------{CMBN 4}
<4 * G4G0>
<-10 * G4G1>
------------------------------------------------------------------------------
<BLANKLINE>
>>> opps_idx2.evaluation(Integer(4), [Integer(2), 'G4G0', -Integer(5), 'G4G1
↪'])
<BLANKLINE>
----------------------------------------------------------------------{CMBN 4}
<-4 * G4G0>
<10 * G4G1>
------------------------------------------------------------------------------
<BLANKLINE>
```

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: # optional - kenzo
sage: from sage.interfaces.kenzo import KChainComplex
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)
sage: kenzo_chcm = KChainComplex(sage_chcm)
sage: kenzo_chcm
[K... Chain-Complex]
sage: idnt = kenzo_chcm.identity_morphism()
sage: idnt
[K... Morphism (degree 0): K... -> K...]
sage: opps_id = idnt.opposite()
sage: opps_id
[K... Morphism (degree 0): K... -> K...]
sage: null = kenzo_chcm.null_morphism()
sage: null
[K... Morphism (degree 0): K... -> K...]
sage: idx2 = idnt.sum(idnt)
sage: idx5 = idx2.sum(
....:   (opps_id, idnt, idnt, null, idx2.sum(idnt), opps_id))
sage: kenzo_chcm.basis(4)
['G4G0', 'G4G1']
```

(continues on next page)

(continued from previous page)

```
sage: idx2.evaluation(4, [2, 'G4G0', -5, 'G4G1'])

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

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

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

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import KChainComplex
>>> m1 = matrix(ZZ, Integer(3), Integer(2), [-Integer(1), Integer(1),␣
↪Integer(3), -Integer(4), Integer(5), Integer(6)])
>>> m4 = matrix(ZZ, Integer(2), Integer(2), [Integer(1), Integer(2),␣
↪Integer(3), Integer(6)])
>>> m5 = matrix(ZZ, Integer(2), Integer(3), [Integer(2), Integer(2),␣
↪Integer(2), -Integer(1), -Integer(1), -Integer(1)])
>>> sage_chcm = ChainComplex({Integer(1): m1, Integer(4): m4, Integer(5): m5},
↪ degree = -Integer(1))
>>> kenzo_chcm = KChainComplex(sage_chcm)
>>> kenzo_chcm
[K... Chain-Complex]
>>> idnt = kenzo_chcm.identity_morphism()
>>> idnt
[K... Morphism (degree 0): K... -> K...]
>>> opps_id = idnt.opposite()
>>> opps_id
[K... Morphism (degree 0): K... -> K...]
>>> null = kenzo_chcm.null_morphism()
>>> null
[K... Morphism (degree 0): K... -> K...]
>>> idx2 = idnt.sum(idnt)
>>> idx5 = idx2.sum(
...   (opps_id, idnt, idnt, null, idx2.sum(idnt), opps_id))
>>> kenzo_chcm.basis(Integer(4))
['G4G0', 'G4G1']
>>> idx2.evaluation(Integer(4), [Integer(2), 'G4G0', -Integer(5), 'G4G1'])
<BLANKLINE>
----------------------------------------------------------------------{CMBN 4}
<4 * G4G0>
<-10 * G4G1>
------------------------------------------------------------------------------
<BLANKLINE>
>>> idx5.evaluation(Integer(4), [Integer(2), 'G4G0', -Integer(5), 'G4G1'])
<BLANKLINE>
----------------------------------------------------------------------{CMBN 4}
```

(continues on next page)

(continued from previous page)

```
<10 * G4G0>
<-25 * G4G1>
------------------------------------------------------------------------------
<BLANKLINE>
```

target_complex()

Return the target chain complex of the morphism.

OUTPUT: a *KenzoChainComplex*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import KChainComplex
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)
sage: kenzo_chcm = KChainComplex(sage_chcm)
sage: kenzo_chcm
[K... Chain-Complex]
sage: differential_morphism = kenzo_chcm.differential()
sage: differential_morphism
[K... Morphism (degree -1): K... -> K...]
sage: differential_morphism.target_complex()
[K... Chain-Complex]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import KChainComplex
>>> m1 = matrix(ZZ, Integer(3), Integer(2), [-Integer(1), Integer(1),␣
↪Integer(3), -Integer(4), Integer(5), Integer(6)])
>>> m4 = matrix(ZZ, Integer(2), Integer(2), [Integer(1), Integer(2),␣
↪Integer(3), Integer(6)])
>>> m5 = matrix(ZZ, Integer(2), Integer(3), [Integer(2), Integer(2),␣
↪Integer(2), -Integer(1), -Integer(1), -Integer(1)])
>>> sage_chcm = ChainComplex({Integer(1): m1, Integer(4): m4, Integer(5): m5},
↪ degree = -Integer(1))
>>> kenzo_chcm = KChainComplex(sage_chcm)
>>> kenzo_chcm
[K... Chain-Complex]
>>> differential_morphism = kenzo_chcm.differential()
>>> differential_morphism
[K... Morphism (degree -1): K... -> K...]
>>> differential_morphism.target_complex()
[K... Chain-Complex]
```

class sage.interfaces.kenzo.**KenzoObject**(*kenzo_object*)

Bases: SageObject

Wrapper to Kenzo objects.

INPUT:

- kenzo_object – a wrapper around a Kenzo object (which is an ecl object)

class sage.interfaces.kenzo.**KenzoSimplicialGroup**(*kenzo_object*)

Bases: *KenzoSimplicialSet*

Wrapper around Kenzo simplicial groups.

classifying_space()

Return the classifying space.

OUTPUT: a *KenzoSimplicialGroup*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import MooreSpace
sage: m2 = MooreSpace(2,4)
sage: l2 = m2.loop_space()
sage: c = l2.classifying_space()
sage: type(c)
<class 'sage.interfaces.kenzo.KenzoSimplicialGroup'>
sage: [c.homology(i) for i in range(8)]
[Z, 0, 0, 0, C2, 0, 0, 0]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import MooreSpace
>>> m2 = MooreSpace(Integer(2),Integer(4))
>>> l2 = m2.loop_space()
>>> c = l2.classifying_space()
>>> type(c)
<class 'sage.interfaces.kenzo.KenzoSimplicialGroup'>
>>> [c.homology(i) for i in range(Integer(8))]
[Z, 0, 0, 0, C2, 0, 0, 0]
```

class sage.interfaces.kenzo.**KenzoSimplicialSet**(*kenzo_object*)

Bases: *KenzoChainComplex*

Wrapper to Kenzo simplicial sets.

In Kenzo, the homology of a simplicial set in computed from its associated chain complex. Hence, this class inherits from $KenzoChainComplex$.

cartesian_product(*other*)

Return the cartesian product of `self` and `other`.

INPUT:

- `other` – the Kenzo simplicial set with which the product is made

OUTPUT: a *KenzoSimplicialSet*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: s2 = Sphere(2)
sage: s3 = Sphere(3)
sage: p = s2.cartesian_product(s3)
sage: type(p)
```

(continues on next page)

(continued from previous page)

```
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
sage: [p.homology(i) for i in range(6)]
[Z, 0, Z, Z, 0, Z]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> s2 = Sphere(Integer(2))
>>> s3 = Sphere(Integer(3))
>>> p = s2.cartesian_product(s3)
>>> type(p)
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
>>> [p.homology(i) for i in range(Integer(6))]
[Z, 0, Z, Z, 0, Z]
```

em_spectral_sequence()

Return the Eilenberg-Moore spectral sequence of `self`.

OUTPUT: a *KenzoSpectralSequence*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: S2 = Sphere(2)
sage: EMS = S2.em_spectral_sequence()
sage: EMS.table(0, -2, 2, -2, 2)
  0   Z   0   0   0
  0   0   0   0   0
  0   0   Z   0   0
  0   0   0   0   0
  0   0   0   0   0
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> S2 = Sphere(Integer(2))
>>> EMS = S2.em_spectral_sequence()
>>> EMS.table(Integer(0), -Integer(2), Integer(2), -Integer(2), Integer(2))
  0   Z   0   0   0
  0   0   0   0   0
  0   0   Z   0   0
  0   0   0   0   0
  0   0   0   0   0
```

> **Warning**
>
> This method assumes that the underlying space is simply connected. You might get wrong answers if it is not.

homotopy_group(n)

Return the n-th homotopy group of `self`.

INPUT:

- `n` – the dimension of the homotopy group to be computed

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: s2 = Sphere(2)
sage: p = s2.cartesian_product(s2)
sage: p.homotopy_group(3)
Multiplicative Abelian group isomorphic to Z x Z
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> s2 = Sphere(Integer(2))
>>> p = s2.cartesian_product(s2)
>>> p.homotopy_group(Integer(3))
Multiplicative Abelian group isomorphic to Z x Z
```

Warning

This method assumes that the underlying space is simply connected. You might get wrong answers if it is not.

join(*other*)

Return the join of `self` and `other`.

INPUT:

- `other` – the Kenzo simplicial set with which the join is made

OUTPUT: a *KenzoSimplicialSet*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: s2 = Sphere(2)
sage: s3 = Sphere(3)
sage: j = s2.join(s3)
sage: type(j)
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
sage: [j.homology(i) for i in range(6)]
[Z, 0, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> s2 = Sphere(Integer(2))
>>> s3 = Sphere(Integer(3))
>>> j = s2.join(s3)
>>> type(j)
```

(continues on next page)

(continued from previous page)

```
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
>>> [j.homology(i) for i in range(Integer(6))]
[Z, 0, 0, 0, 0, 0]
```

loop_space(*n=1*)

Return the n-th iterated loop space.

INPUT:

- n – (default: 1) the number of times to iterate the loop space construction

OUTPUT: a *KenzoSimplicialGroup*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: s2 = Sphere(2)
sage: l2 = s2.loop_space()
sage: type(l2)
<class 'sage.interfaces.kenzo.KenzoSimplicialGroup'>
sage: l2 = s2.loop_space()
sage: [l2.homology(i) for i in range(8)]
[Z, Z, Z, Z, Z, Z, Z, Z]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> s2 = Sphere(Integer(2))
>>> l2 = s2.loop_space()
>>> type(l2)
<class 'sage.interfaces.kenzo.KenzoSimplicialGroup'>
>>> l2 = s2.loop_space()
>>> [l2.homology(i) for i in range(Integer(8))]
[Z, Z, Z, Z, Z, Z, Z, Z]
```

serre_spectral_sequence()

Return the spectral sequence of self.

The object self must be created as a cartesian product (twisted or not).

OUTPUT: a *KenzoSpectralSequence*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: S2 = Sphere(2)
sage: S3 = Sphere(3)
sage: P = S2.cartesian_product(S3)
sage: E = P.serre_spectral_sequence()
sage: E.table(0, 0, 2, 0, 3)
  Z   0   Z
  0   0   0
  0   0   0
  Z   0   Z
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> S2 = Sphere(Integer(2))
>>> S3 = Sphere(Integer(3))
>>> P = S2.cartesian_product(S3)
>>> E = P.serre_spectral_sequence()
>>> E.table(Integer(0), Integer(0), Integer(2), Integer(0), Integer(3))
  Z   0   Z
  0   0   0
  0   0   0
  Z   0   Z
```

> **Warning**
>
> This method assumes that the underlying space is simply connected. You might get wrong answers if it is not.

smash_product(*other*)

Return the smash product of `self` and `other`.

INPUT:

- `other` – the Kenzo simplicial set with which the smash product is made

OUTPUT: a *KenzoSimplicialSet*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: s2 = Sphere(2)
sage: s3 = Sphere(3)
sage: s = s2.smash_product(s3)
sage: type(s)
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
sage: [s.homology(i) for i in range(6)]
[Z, 0, 0, 0, 0, Z]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> s2 = Sphere(Integer(2))
>>> s3 = Sphere(Integer(3))
>>> s = s2.smash_product(s3)
>>> type(s)
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
>>> [s.homology(i) for i in range(Integer(6))]
[Z, 0, 0, 0, 0, Z]
```

suspension()

Return the suspension of the simplicial set.

OUTPUT: a *KenzoSimplicialSet*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import EilenbergMacLaneSpace
sage: e3 = EilenbergMacLaneSpace(ZZ, 3)
sage: s = e3.suspension()
sage: type(s)
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
sage: [s.homology(i) for i in range(6)]
[Z, 0, 0, 0, Z, 0]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import EilenbergMacLaneSpace
>>> e3 = EilenbergMacLaneSpace(ZZ, Integer(3))
>>> s = e3.suspension()
>>> type(s)
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
>>> [s.homology(i) for i in range(Integer(6))]
[Z, 0, 0, 0, Z, 0]
```

sw_spectral_sequence()

Return the Serre sequence of the first step of the Whitehead tower.

OUTPUT: a *KenzoSpectralSequence*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: S3 = Sphere(3)
sage: E = S3.sw_spectral_sequence()
sage: T = E.table(0, 0, 4, 0, 4)
sage: T
  Z    0    0    Z    0
  0    0    0    0    0
  Z    0    0    Z    0
  0    0    0    0    0
  Z    0    0    Z    0
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> S3 = Sphere(Integer(3))
>>> E = S3.sw_spectral_sequence()
>>> T = E.table(Integer(0), Integer(0), Integer(4), Integer(0), Integer(4))
>>> T
  Z    0    0    Z    0
  0    0    0    0    0
  Z    0    0    Z    0
  0    0    0    0    0
  Z    0    0    Z    0
```

wedge(*other*)

Return the wedge of `self` and `other`.

INPUT:

- other – the Kenzo simplicial set with which the wedge is made

OUTPUT: a *KenzoSimplicialSet*

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: s2 = Sphere(2)
sage: s3 = Sphere(3)
sage: w = s2.wedge(s3)
sage: type(w)
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
sage: [w.homology(i) for i in range(6)]
[Z, 0, Z, Z, 0, 0]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> s2 = Sphere(Integer(2))
>>> s3 = Sphere(Integer(3))
>>> w = s2.wedge(s3)
>>> type(w)
<class 'sage.interfaces.kenzo.KenzoSimplicialSet'>
>>> [w.homology(i) for i in range(Integer(6))]
[Z, 0, Z, Z, 0, 0]
```

class sage.interfaces.kenzo.**KenzoSpectralSequence**(*kenzo_object*)

Bases: *KenzoObject*

Wrapper around Kenzo spectral sequences

differential(*p*, *i*, *j*)

Return the (p, i, j) differential morphism of the spectral sequence.

INPUT:

- p – the page
- i – the column of the differential domain
- j – the row of the differential domain

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: S3 = Sphere(3)
sage: L = S3.loop_space()
sage: EMS = L.em_spectral_sequence()
sage: EMS.table(1,-5,-2,5,8)
  0   Z   Z + Z + Z   Z + Z + Z
  0   0   0           0
  0   0   Z           Z + Z
  0   0   0           0
sage: EMS.matrix(1, -3, 8)
```

(continues on next page)

(continued from previous page)

```
[ 2 -2  2]
sage: EMS.differential(1, -3, 8)
Morphism from module over Integer Ring with invariants (0, 0, 0) to module␣
↪with invariants (0,) that sends the generators to [(2), (-2), (2)]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> S3 = Sphere(Integer(3))
>>> L = S3.loop_space()
>>> EMS = L.em_spectral_sequence()
>>> EMS.table(Integer(1),-Integer(5),-Integer(2),Integer(5),Integer(8))
  0   Z   Z + Z + Z   Z + Z + Z
  0   0   0           0
  0   0   Z           Z + Z
  0   0   0           0
>>> EMS.matrix(Integer(1), -Integer(3), Integer(8))
[ 2 -2  2]
>>> EMS.differential(Integer(1), -Integer(3), Integer(8))
Morphism from module over Integer Ring with invariants (0, 0, 0) to module␣
↪with invariants (0,) that sends the generators to [(2), (-2), (2)]
```

group (p, i, j)

Return the `i,j`-th group of the `p` page.

INPUT:

- `p` – the page to take the group from
- `i` – the column where the group is taken from
- `j` – the row where the group is taken from

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: S2 = Sphere(2)
sage: EMS = S2.em_spectral_sequence()
sage: EMS.group(0, -1, 2)
Additive abelian group isomorphic to Z
sage: EMS.group(0, -1, 3)
Trivial group
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> S2 = Sphere(Integer(2))
>>> EMS = S2.em_spectral_sequence()
>>> EMS.group(Integer(0), -Integer(1), Integer(2))
Additive abelian group isomorphic to Z
>>> EMS.group(Integer(0), -Integer(1), Integer(3))
Trivial group
```

matrix(*p*, *i*, *j*)

Return the matrix that determines the differential from the `i,j`-th group of the `p`-th page.

INPUT:

- `p` – the page
- `i` – the column of the differential domain
- `j` – the row of the differential domain

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: S3 = Sphere(3)
sage: L = S3.loop_space()
sage: EMS = L.em_spectral_sequence()
sage: EMS.table(1, -5, -2, 5, 8)
  0   Z   Z + Z + Z   Z + Z + Z
  0   0   0           0
  0   0   Z           Z + Z
  0   0   0           0
sage: EMS.matrix(1, -2 ,8)
[ 3 -2  0]
[ 3  0 -3]
[ 0  2 -3]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> S3 = Sphere(Integer(3))
>>> L = S3.loop_space()
>>> EMS = L.em_spectral_sequence()
>>> EMS.table(Integer(1), -Integer(5), -Integer(2), Integer(5), Integer(8))
  0   Z   Z + Z + Z   Z + Z + Z
  0   0   0           0
  0   0   Z           Z + Z
  0   0   0           0
>>> EMS.matrix(Integer(1), -Integer(2) ,Integer(8))
[ 3 -2  0]
[ 3  0 -3]
[ 0  2 -3]
```

table(*p*, *i1*, *i2*, *j1*, *j2*)

Return a table printing the groups in the `p` page.

INPUT:

- `p` – the page to print

– `i1` – the first column to print

– `i2` – the last column to print

– `j1` – the first row to print

– `j2` – the last row to print

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: S2 = Sphere(2)
sage: EMS = S2.em_spectral_sequence()
sage: EMS.table(0, -2, 2, -2, 2)
  0   Z   0   0   0
  0   0   0   0   0
  0   0   Z   0   0
  0   0   0   0   0
  0   0   0   0   0
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> S2 = Sphere(Integer(2))
>>> EMS = S2.em_spectral_sequence()
>>> EMS.table(Integer(0), -Integer(2), Integer(2), -Integer(2), Integer(2))
  0   Z   0   0   0
  0   0   0   0   0
  0   0   Z   0   0
  0   0   0   0   0
  0   0   0   0   0
```

sage.interfaces.kenzo.**MooreSpace**(*m*, *n*)

Return the Moore space $M(m, n)$ as a Kenzo simplicial set.

The Moore space $M(m, n)$ is the space whose n-th homology group is isomorphic to the cyclic group of order m, and the rest of the homology groups are trivial.

INPUT:

- m – positive integer; the order of the nontrivial homology group
- n – the dimension in which the homology is not trivial

OUTPUT: a KenzoSimplicialSet

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import MooreSpace
sage: m24 = MooreSpace(2,4)
sage: m24
[K10 Simplicial-Set]
sage: [m24.homology(i) for i in range(8)]
[Z, 0, 0, 0, C2, 0, 0, 0]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import MooreSpace
>>> m24 = MooreSpace(Integer(2),Integer(4))
>>> m24
[K10 Simplicial-Set]
>>> [m24.homology(i) for i in range(Integer(8))]
[Z, 0, 0, 0, C2, 0, 0, 0]
```

sage.interfaces.kenzo.**SAbstractSimplex**(*simplex*, *dim*)

Convert an abstract simplex of Kenzo to an AbstractSimplex.

INPUT:

- `simplex` – an abstract simplex of Kenzo
- `dim` – the dimension of `simplex`

OUTPUT: an AbstractSimplex

EXAMPLES:

```
sage: # optional - kenzo
sage: from sage.libs.ecl import EclObject, ecl_eval
sage: from sage.interfaces.kenzo import (
....:   KenzoObject, SAbstractSimplex)
sage: KAbSm = KenzoObject(ecl_eval("(ABSM 15 'K)"))
sage: SAbSm1 = SAbstractSimplex(KAbSm, 2)
sage: SAbSm2 = SAbstractSimplex(KAbSm, 7)
sage: SAbSm1.degeneracies()
[3, 2, 1, 0]
sage: SAbSm1.dimension()
6
sage: SAbSm2.dimension()
11
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.libs.ecl import EclObject, ecl_eval
>>> from sage.interfaces.kenzo import (
...   KenzoObject, SAbstractSimplex)
>>> KAbSm = KenzoObject(ecl_eval("(ABSM 15 'K)"))
>>> SAbSm1 = SAbstractSimplex(KAbSm, Integer(2))
>>> SAbSm2 = SAbstractSimplex(KAbSm, Integer(7))
>>> SAbSm1.degeneracies()
[3, 2, 1, 0]
>>> SAbSm1.dimension()
6
>>> SAbSm2.dimension()
11
```

sage.interfaces.kenzo.**SChainComplex**(*kchaincomplex*, *start=0*, *end=15*)

Convert the KenzoChainComplex `kchcm` (between dimensions `start` and `end`) to a ChainComplex.

INPUT:

- `kchaincomplex` – a KenzoChainComplex
- `start` – integer (default: 0)
- `end` – integer greater than or equal to `start` (default: 15)

OUTPUT: a ChainComplex

EXAMPLES:

```
sage: from sage.interfaces.kenzo import KChainComplex, SChainComplex   # 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: SChainComplex(KChainComplex(sage_chcm)) == sage_chcm             # optional␣
↪- kenzo
True
```

```
>>> from sage.all import *
>>> from sage.interfaces.kenzo import KChainComplex, SChainComplex   # optional -␣
↪kenzo
>>> m1 = matrix(ZZ, Integer(3), Integer(2), [-Integer(1), Integer(1), Integer(3),␣
↪-Integer(4), Integer(5), Integer(6)])
>>> m4 = matrix(ZZ, Integer(2), Integer(2), [Integer(1), Integer(2), Integer(3),␣
↪Integer(6)])
>>> m5 = matrix(ZZ, Integer(2), Integer(3), [Integer(2), Integer(2), Integer(2), -
↪Integer(1), -Integer(1), -Integer(1)])
>>> sage_chcm = ChainComplex({Integer(1): m1, Integer(4): m4, Integer(5): m5},␣
↪degree = -Integer(1))     # optional - kenzo
>>> SChainComplex(KChainComplex(sage_chcm)) == sage_chcm             # optional -␣
↪kenzo
True
```

```
sage: # optional - kenzo
sage: from sage.interfaces.kenzo import SChainComplex, Sphere
sage: S4 = Sphere(4)
sage: C = SChainComplex(S4)
sage: C
Chain complex with at most 3 nonzero terms over Integer Ring
sage: C._ascii_art_()
0 <-- C_4 <-- 0  ...  0 <-- C_0 <-- 0
sage: [C.homology(i) for i in range(6)]
[Z, 0, 0, 0, Z, 0]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import SChainComplex, Sphere
>>> S4 = Sphere(Integer(4))
>>> C = SChainComplex(S4)
>>> C
Chain complex with at most 3 nonzero terms over Integer Ring
>>> C._ascii_art_()
0 <-- C_4 <-- 0  ...  0 <-- C_0 <-- 0
>>> [C.homology(i) for i in range(Integer(6))]
[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: # optional - kenzo
sage: from sage.topology.simplicial_set import SimplicialSet
sage: from sage.interfaces.kenzo import (
....:   AbstractSimplex, KFiniteSimplicialSet,
....:   SFiniteSimplicialSet, Sphere)
sage: s0 = AbstractSimplex(0, name='s0')
sage: s1 = AbstractSimplex(0, name='s1')
sage: s2 = AbstractSimplex(0, name='s2')
sage: s01 = AbstractSimplex(1, name='s01')
sage: s02 = AbstractSimplex(1, name='s02')
sage: s12 = AbstractSimplex(1, name='s12')
sage: s012 = AbstractSimplex(2, name='s012')
sage: Triangle = SimplicialSet({s01: (s1, s0),
....:                            s02: (s2, s0),
....:                            s12: (s2, s1)},
....:                           base_point = s0)
sage: KTriangle = KFiniteSimplicialSet(Triangle)
sage: STriangle = SFiniteSimplicialSet(KTriangle, 1)
sage: STriangle.homology()
{0: 0, 1: Z}
sage: S1 = simplicial_sets.Sphere(1)
sage: S3 = simplicial_sets.Sphere(3)
sage: KS1vS3 = KFiniteSimplicialSet(S1.wedge(S3))
sage: SS1vS3 = SFiniteSimplicialSet(KS1vS3, 3)
sage: SS1vS3.homology()
{0: 0, 1: Z, 2: 0, 3: Z}
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.topology.simplicial_set import SimplicialSet
>>> from sage.interfaces.kenzo import (
...   AbstractSimplex, KFiniteSimplicialSet,
...   SFiniteSimplicialSet, Sphere)
>>> s0 = AbstractSimplex(Integer(0), name='s0')
>>> s1 = AbstractSimplex(Integer(0), name='s1')
>>> s2 = AbstractSimplex(Integer(0), name='s2')
>>> s01 = AbstractSimplex(Integer(1), name='s01')
>>> s02 = AbstractSimplex(Integer(1), name='s02')
>>> s12 = AbstractSimplex(Integer(1), name='s12')
>>> s012 = AbstractSimplex(Integer(2), name='s012')
>>> Triangle = SimplicialSet({s01: (s1, s0),
...                            s02: (s2, s0),
...                            s12: (s2, s1)},
...                           base_point = s0)
>>> KTriangle = KFiniteSimplicialSet(Triangle)
>>> STriangle = SFiniteSimplicialSet(KTriangle, Integer(1))
>>> STriangle.homology()
```

(continues on next page)

(continued from previous page)

```
{0: 0, 1: Z}
>>> S1 = simplicial_sets.Sphere(Integer(1))
>>> S3 = simplicial_sets.Sphere(Integer(3))
>>> KS1vS3 = KFiniteSimplicialSet(S1.wedge(S3))
>>> SS1vS3 = SFiniteSimplicialSet(KS1vS3, Integer(3))
>>> SS1vS3.homology()
{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: # optional - kenzo
sage: from sage.interfaces.kenzo import Sphere
sage: s2 = Sphere(2)
sage: s2
[K1 Simplicial-Set]
sage: [s2.homology(i) for i in range(8)]
[Z, 0, Z, 0, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import Sphere
>>> s2 = Sphere(Integer(2))
>>> s2
[K1 Simplicial-Set]
>>> [s2.homology(i) for i in range(Integer(8))]
[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` – integer 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` – list containing a description about the origin of the morphism

OUTPUT: a *KenzoChainComplexMorphism*

EXAMPLES:

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

```
>>> from sage.all import *
>>> # optional - kenzo
>>> from sage.interfaces.kenzo import (KenzoChainComplex,
...                                    build_morphism)
>>> from sage.libs.ecl import ecl_eval
>>> ZCC = KenzoChainComplex(ecl_eval("(z-chcm)"))
>>> A = build_morphism(
...   ZCC, ZCC, -Integer(1),
...   ecl_eval("#'(lambda (comb) (cmbn (1- (degr comb))))"),
...   "cmbn", ["zero morphism on ZCC"])
>>> A.target_complex()
[K... Chain-Complex]
>>> A.degree()
-1
>>> type(A)
<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
[1 2 3]
[3 2 1]
[1 1 1]
```

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

(continues on next page)

(continued from previous page)

```
>>> k2s_matrix(M)                                          # optional - kenzo
[1 2 3]
[3 2 1]
[1 1 1]
```

sage.interfaces.kenzo.**morphism_dictmat**(*morphism*)

Compute 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)))))>
```

```
>>> from sage.all import *
>>> from sage.interfaces.kenzo import morphism_dictmat    # optional - kenzo
>>> X = simplicial_complexes.Simplex(Integer(1))
>>> Y = simplicial_complexes.Simplex(Integer(0))
>>> g = Hom(X,Y)({Integer(0):Integer(0), Integer(1):Integer(0)})
>>> f = g.associated_chain_complex_morphism()
>>> 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` – 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))>
```

```
>>> from sage.all import *
>>> from sage.interfaces.kenzo import pairing   # optional - kenzo
>>> l = [Integer(1),Integer(2),Integer(3)]
>>> 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` – 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}
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))))>
```

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

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

INPUT:

- `l` – 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...])>
```

```
>>> from sage.all import *
>>> from sage.interfaces.kenzo import s2k_listofmorphisms   # optional - kenzo
>>> C1 = ChainComplex({Integer(1): matrix(ZZ, Integer(0), Integer(2), [])},␣
↪degree_of_differential=-Integer(1))
>>> C2 = ChainComplex({Integer(1): matrix(ZZ, Integer(1), Integer(2), [Integer(1),
↪ Integer(0)])},degree_of_differential=-Integer(1))
>>> C3 = ChainComplex({Integer(0): matrix(ZZ, Integer(0),Integer(2) , [])},degree_
```

(continues on next page)

(continued from previous page)

```
↪of_differential=-Integer(1))
>>> M1 = Hom(C2,C1)({Integer(1): matrix(ZZ, Integer(2), Integer(2), [Integer(2),␣
↪Integer(0), Integer(0), Integer(2)])})
>>> M2 = Hom(C3,C2)({Integer(0): matrix(ZZ, Integer(1), Integer(2), [Integer(2),␣
↪Integer(0)])})
>>> l = [M1, M2]
>>> 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))>
```

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

CHAPTER

NINETEEN

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()
```

```
>>> from sage.all import *
>>> from sage.interfaces.latte import count     # optional - latte_int
>>> P = Integer(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
```

```
>>> from sage.all import *
>>> count(P.cdd_Hrepresentation(), cdd=True)    # optional - latte_int
125
>>> 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
```

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

```
>>> from sage.all import *
>>> opts = {'cdd': True,
...         'multivariate_generating_function': True,
...         'raw_output': True}
>>> cddin = P.cdd_Hrepresentation()
>>> 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')
```

```
>>> from sage.all import *
>>> from sage.interfaces.latte import integrate   # optional - latte_int
>>> P = Integer(2) * polytopes.cube()
>>> 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
```

```
>>> from sage.all import *
>>> integrate(P.cdd_Hrepresentation(), x**Integer(2)*y**Integer(2)*z**Integer(2),
↪cdd=True)   # optional - latte_int
4096/27
>>> integrate(P.cdd_Vrepresentation(), x**Integer(2)*y**Integer(2)*z**Integer(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
```

```
>>> from sage.all import *
>>> integrate(P.cdd_Hrepresentation(), cdd=True)   # optional - latte_int
64
>>> 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
```

```
>>> from sage.all import *
>>> 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: string that describes the monomials list and exponent vectors

CHAPTER

TWENTY

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
O---O---O---O
1   2   3   4
A4

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

sage: a4.diagram()               # optional - lie
O---O---O---O
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]
```

```
>>> from sage.all import *
>>> a4 = lie('A4')             # optional - lie
>>> lie.diagram('A4')          # optional - lie
O---O---O---O
```

(continues on next page)

(continued from previous page)

```
1   2   3   4
A4

>>> lie.diagram(a4)              # optional - lie
O---O---O---O
1   2   3   4
A4

>>> a4.diagram()                 # optional - lie
O---O---O---O
1   2   3   4
A4

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

20.1 Tutorial

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

You can perform basic arithmetic operations in LiE.

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

```
>>> from sage.all import *
>>> # optional - lie
>>> lie.eval('19+68')
'87'
>>> a = lie('1111111111*1111111111')
>>> a
1234567900987654321
>>> a/Integer(1111111111)
1111111111
```

(continues on next page)

(continued from previous page)

```
>>> a = lie('345')
>>> a**Integer(2)+Integer(3)*a-Integer(5)
120055
>>> _ / Integer(7)*a
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: # optional - lie
sage: v = lie('[3,2,6873,-38]')
sage: v
[3,2,6873,-38]
sage: v[3]
6873
sage: v+v
[6,4,13746,-76]
sage: v*v
47239586
sage: v+234786
[3,2,6873,-38,234786]
sage: v-3
[3,2,-38]
sage: v^v
[3,2,6873,-38,3,2,6873,-38]
```

```
>>> from sage.all import *
>>> # optional - lie
>>> v = lie('[3,2,6873,-38]')
>>> v
[3,2,6873,-38]
>>> v[Integer(3)]
6873
>>> v+v
[6,4,13746,-76]
>>> v*v
47239586
>>> v+Integer(234786)
[3,2,6873,-38,234786]
>>> v-Integer(3)
[3,2,-38]
>>> v**v
[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]
```

(continues on next page)

(continued from previous page)

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

sage: # optional - lie
sage: m^3
      [[ 220,   87, 81, 375]
      ,[-168,-1089, 13,1013]
      ,[1550,  357,-55,1593]
      ,[-854, -652, 98,-170]
      ]
sage: v*m
[-6960,62055,55061,-319]
sage: m*v
[20508,-27714,54999,-14089]
sage: v*m*v
378549605
sage: m+v
      [[ 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]
      ]
```

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

>>> # optional - lie
>>> m**Integer(3)
```

(continues on next page)

(continued from previous page)

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

>>> m-Integer(2) # optional - lie
     [[ 1, 0, 3,3]
     ,[-1, 9, 8,0]
     ,[ 3,-5,-2,9]
     ]
```

LiE handles multivariate (Laurent) polynomials.

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

```
>>> from sage.all import *
>>> # optional - lie
>>> lie('X[1,2]')
1X[1,2]
>>> -Integer(3)*_
-3X[1,2]
>>> _ + lie('4X[-1,4]')
4X[-1,4] - 3X[ 1,2]
>>> _**Integer(2)
16X[-2,8] - 24X[ 0,6] +  9X[ 2,4]
>>> lie('(4X[-1,4]-3X[1,2])*(X[2,0]-X[0,-4])')
```

(continues on next page)

(continued from previous page)

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

```
>>> from sage.all import *
>>> lie.partitions(Integer(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]
     ,[2,2,1,1,0,0]
     ,[2,1,1,1,1,0]
     ,[1,1,1,1,1,1]
     ]
>>> lie.diagram('E8') # optional - lie
        O 2
        |
        |
O---O---O---O---O---O---O
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: # optional - lie
sage: lie.eval('f(int x) = 2*x')
''
sage: lie.f(984)
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')
''
sage: lie.f(2)
-53
sage: lie.f(5)
5765224
```

```
>>> from sage.all import *
>>> # optional - lie
>>> lie.eval('f(int x) = 2*x')
''
>>> lie.f(Integer(984))
1968
>>> lie.eval('f(int n) = a=3*n-7; if a < 0 then a = -a fi; 7^a+a^3-4*a-57')
''
>>> lie.f(Integer(2))
-53
>>> lie.f(Integer(5))
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.
```

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

(continues on next page)

(continued from previous page)

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

```
>>> from sage.all import *
>>> a = lie('1234') # optional - lie
>>> b = a.sage(); b # optional - lie
1234
>>> 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'>
```

```
>>> from sage.all import *
>>> a = lie('[1,2,3]') # optional - lie
>>> b = a.sage(); b # optional - lie
[1, 2, 3]
>>> 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'>
```

```
>>> from sage.all import *
>>> a = lie('[[1,2],[3,4]]') # optional - lie
>>> b = a.sage(); b # optional - lie
[1 2]
[3 4]
>>> 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'>
```

```
>>> from sage.all import *
>>> a = lie('X[1,2] - 2*X[2,1]') # optional - lie
>>> b = a.sage(); b              # optional - lie
-2*x0^2*x1 + x0*x1^2
>>> 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'>
```

```
>>> from sage.all import *
>>> a = lie('"text"') # optional - lie
>>> b = a.sage(); b # optional - lie
'text'
>>> 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: # optional - lie
sage: def eval_pol(p, pt):
....:     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]')
sage: b1 = lie('[1,2]')
sage: b2 = lie('[2,3]')
sage: eval_pol(a, b1)
4
sage: eval_pol(a, b2)
18
```

```
>>> from sage.all import *
>>> # optional - lie
>>> def eval_pol(p, pt):
...     s = Integer(0)
...     for i in range(Integer(1),p.length().sage()+Integer(1)):
...         m = Integer(1)
...         for j in range(Integer(1),pt.size().sage()+Integer(1)):
...             m *= pt[j]**p.expon(i)[j]
...         s += p.coef(i)*m
...     return s
```

(continues on next page)

(continued from previous page)

```
>>> a = lie('X[1,2]')
>>> b1 = lie('[1,2]')
>>> b2 = lie('[2,3]')
>>> eval_pol(a, b1)
4
>>> eval_pol(a, b2)
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: `ExtraTabCompletion`, *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
...
```

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

```
>>> from sage.all import *
>>> lie.eval('2+2')  # optional - lie
'4'
```

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

EXAMPLES:

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

```
>>> from sage.all import *
>>> lie.function_call("diagram", args=['A4']) # optional - lie
O---O---O---O
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'
```

```
>>> from sage.all import *
>>> lie.set('x', '2')  # optional - lie
>>> lie.get('x')       # optional - lie
'2'
```

get_using_file(*var*)

EXAMPLES:

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

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

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

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

```
>>> from sage.all import *
>>> lie.set('x', '2')  # optional - lie
>>> lie.get('x')       # optional - lie
'2'
```

version()

EXAMPLES:

```
sage: lie.version() # optional - lie
'2...'
```

```
>>> from sage.all import *
>>> lie.version() # optional - lie
'2...'
```

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

Bases: `ExtraTabCompletion`, *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'
```

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

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

Bases: *ExpectFunction*

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

Bases: *FunctionElement*

sage.interfaces.lie.**is_LiEElement**(*x*)

EXAMPLES:

```
sage: from sage.interfaces.lie import is_LiEElement
sage: is_LiEElement(2)
doctest:...: DeprecationWarning: the function is_LiEElement is deprecated; use
↪isinstance(x, sage.interfaces.abc.LiEElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
sage: l = lie(2) # optional - lie
sage: is_LiEElement(l) # optional - lie
True
```

```
>>> from sage.all import *
>>> from sage.interfaces.lie import is_LiEElement
>>> is_LiEElement(Integer(2))
doctest:...: DeprecationWarning: the function is_LiEElement is deprecated; use
↪isinstance(x, sage.interfaces.abc.LiEElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
>>> l = lie(Integer(2)) # optional - lie
>>> is_LiEElement(l) # optional - lie
True
```

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
...
```

```
>>> from sage.all import *
>>> from sage.interfaces.lie import lie_console
>>> 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...'
```

```
>>> from sage.all import *
>>> from sage.interfaces.lie import lie_version
>>> lie_version() # optional - lie
'2...'
```

sage.interfaces.lie.**reduce_load_lie**()

EXAMPLES:

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

```
>>> from sage.all import *
>>> from sage.interfaces.lie import reduce_load_lie
>>> reduce_load_lie()
LiE Interpreter
```

CHAPTER

TWENTYONE

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'
```

```
>>> from sage.all import *
>>> lisp.eval('(* 4 5)')
'20'
>>> a = lisp(Integer(3)); b = lisp(Integer(5))
>>> a + b
8
>>> a * b
15
>>> a / b
3/5
>>> a - b
-2
>>> a.sin()
0.14112
>>> b.cos()
0.2836622
>>> a.exp()
20.085537
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

```
>>> from sage.all import *
>>> lisp.eval('(defun factorial (n) (if (= n 1) 1 (* n (factorial (- n 1)))))')
'FACTORIAL'
>>> lisp('(factorial 10)')
3628800
>>> lisp(Integer(10)).factorial()
3628800
>>> a = lisp(Integer(17))
>>> 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: *Expect*

EXAMPLES:

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

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

(continues on next page)

(continued from previous page)

```
under certain conditions; see file 'Copyright' for details.
Type :h for Help.  Top level.
...
```

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

```
>>> from sage.all import *
>>> lisp.eval('(+ 2 2)')
'4'
```

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

Call 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
```

```
>>> from sage.all import *
>>> lisp.function_call('sin', ['2'])
0.9092974
>>> lisp.sin(Integer(2))
0.9092974
```

get(*var*)

EXAMPLES:

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

```
>>> from sage.all import *
>>> lisp.set('x', '2')
>>> lisp.get('x')
'2'
```

help(*command*)

EXAMPLES:

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

```
>>> from sage.all import *
>>> lisp.help('setq')
Traceback (most recent call last):
...
NotImplementedError
```

kill(*var*)

EXAMPLES:

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

```
>>> from sage.all import *
>>> lisp.kill('x')
Traceback (most recent call last):
...
NotImplementedError
```

set(*var*, *value*)

Set the variable var to the given value.

EXAMPLES:

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

```
>>> from sage.all import *
>>> lisp.set('x', '2')
>>> lisp.get('x')
'2'
```

version()

Return the version of Lisp being used.

EXAMPLES:

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

```
>>> from sage.all import *
>>> lisp.version()
'Version information is given by lisp.console().'
```

class sage.interfaces.lisp.**LispElement**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: RingElement, *ExpectElement*

class sage.interfaces.lisp.**LispFunction**(*parent*, *name*)

Bases: *ExpectFunction*

class sage.interfaces.lisp.**LispFunctionElement**(*obj*, *name*)

Bases: *FunctionElement*

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

EXAMPLES:

```
sage: from sage.interfaces.lisp import is_LispElement
sage: is_LispElement(2)
doctest:...: DeprecationWarning: the function is_LispElement is deprecated; use␣
↪isinstance(x, sage.interfaces.abc.LispElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
sage: is_LispElement(lisp(2))
True
```

```
>>> from sage.all import *
>>> from sage.interfaces.lisp import is_LispElement
>>> is_LispElement(Integer(2))
doctest:...: DeprecationWarning: the function is_LispElement is deprecated; use␣
↪isinstance(x, sage.interfaces.abc.LispElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
>>> is_LispElement(lisp(Integer(2)))
True
```

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.
...
```

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

(continues on next page)

(continued from previous page)

```
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
```

```
>>> from sage.all import *
>>> from sage.interfaces.lisp import reduce_load_Lisp
>>> reduce_load_Lisp()
Lisp Interpreter
```

CHAPTER

TWENTYTWO

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

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

(continues on next page)

(continued from previous page)

```
x   + y
sage: parent((x+y)^5)
Macaulay2
```

```
>>> from sage.all import *
>>> # optional - macaulay2
>>> macaulay2('3/5 + 7/11')
68
--
55
>>> f = macaulay2('f = i -> i^3')
>>> f
f
>>> f(Integer(5))
125

>>> # optional - macaulay2
>>> R = macaulay2('ZZ/5[x,y,z]')
>>> R
ZZ
--[x...z]
 5
>>> x = macaulay2('x')
>>> y = macaulay2('y')
>>> (x+y)**Integer(5)
 5    5
x   + y
>>> parent((x+y)**Integer(5))
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: # optional - macaulay2
sage: R = macaulay2('QQ[x,y,z,w]', 'R')
sage: R
R

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

```
>>> from sage.all import *
>>> # optional - macaulay2
```

(continues on next page)

(continued from previous page)

```
>>> R = macaulay2('QQ[x,y,z,w]', 'R')
>>> R
R

>>> # optional - macaulay2
>>> f = macaulay2('x^4 + 2*x*y^3 + x*y^2*w + x*y*z*w + x*y*w^2'
...               '+ 2*x*z*w^2 + y^4 + y^3*w + 2*y^2*z*w + z^4 + w^4')
>>> f
 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
>>> g = f * macaulay2('x+y^5')
>>> print(g.factor())
  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: # optional - macaulay2
sage: macaulay2.eval('compactMatrixForm')
true
sage: macaulay2.eval('compactMatrixForm = false;')
sage: macaulay2.eval('matrix {{1, x^2+y}}')
|     2     |
| 1  x  + y |

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

```
>>> from sage.all import *
>>> # optional - macaulay2
>>> macaulay2.eval('compactMatrixForm')
true
>>> macaulay2.eval('compactMatrixForm = false;')
>>> macaulay2.eval('matrix {{1, x^2+y}}')
|     2     |
| 1  x  + y |
<BLANKLINE>
        1       2
Matrix R  <--- R
>>> macaulay2.eval('compactMatrixForm = true;')
```

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

```
>>> from sage.all import *
>>> # optional - macaulay2
>>> macaulay2.eval('R = QQ[x,y];')
>>> macaulay2.eval('net class R')
PolynomialRing
>>> macaulay2.clear('R')
>>> macaulay2.eval('net class R')
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, LLLBases, Parsing,
↪ PrimaryDecomposition, SchurRings, TangentCone
...
```

```
>>> from sage.all import *
>>> macaulay2.console()                    # not tested
Macaulay 2, version 1.1
with packages: Classic, Core, Elimination, IntegralClosure, LLLBases, Parsing,
↪ PrimaryDecomposition, SchurRings, TangentCone
...
```

cputime(*t=None*)

EXAMPLES:

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

```
>>> from sage.all import *
>>> # optional - macaulay2
>>> R = macaulay2("QQ[x,y]")
>>> x,y = R.gens()
>>> a = (x+y+Integer(1))**Integer(20)
>>> macaulay2.cputime()       # 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` – string
- `strip` – ignored

EXAMPLES:

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

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

```
>>> from sage.all import *
>>> macaulay2.set("a", "2") # optional - macaulay2
>>> 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'>
```

```
>>> from sage.all import *
>>> a = macaulay2('2'); a    # optional - macaulay2
2
>>> 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...
```

```
>>> from sage.all import *
>>> macaulay2.help("load")  # optional - macaulay2 - 1st call might be chatty.
↪..
...
>>> 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
```

(continues on next page)

(continued from previous page)

```
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             |
```

```
>>> from sage.all import *
>>> R2 = macaulay2.ring('QQ', '[x, y]'); R2           # optional - macaulay2
QQ[x...y]
>>> I = macaulay2.ideal( ('y^2 - x^3', 'x - y') ); I    # optional - macaulay2
          3    2
ideal (- x  + y , x - y)
>>> J = I**Integer(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]
```

```
>>> from sage.all import *
>>> l = macaulay2.new_from("MutableList", [Integer(1),Integer(2),Integer(3)])
↪# optional - macaulay2
>>> l                                                # optional - macaulay2
MutableList{...3...}
>>> list(l)                                          # optional - macaulay2
[1, 2, 3]
```

options = Current options for Macaulay2 - after_print: False

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` – 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
```

```
>>> from sage.all import *
>>> R1 = macaulay2.ring('ZZ/7', '[a..d]', 'GRevLex')  # optional - macaulay2
>>> R1.describe()  # optional - macaulay2
ZZ
--[a..d, Degrees => {4:1}, Heft => {1}, MonomialOrder => {MonomialSize => 16},
 7                                                        {GRevLex => {4:1}  }
                                                          {Position => Up    }
--------------------------------------------------------------------------------
↪--
DegreeRank => 1]
>>> 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},
                                                          {Lex => 2          }
                                                          {Position => Up    }
--------------------------------------------------------------------------------
↪--
DegreeRank => 1]
```

```
>>> from sage.all import *
>>> R2 = macaulay2.ring('QQ', '[x, y]')  # optional - macaulay2
>>> R2.describe()  # optional - macaulay2
QQ[x..y, Degrees => {2:1}, Heft => {1}, MonomialOrder => {MonomialSize => 16},
                                                          {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` – string to evaluate

EXAMPLES:

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

```
>>> from sage.all import *
>>> macaulay2.set("a", "1+1")  # optional - macaulay2
>>> 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]
```

```
>>> from sage.all import *
>>> m = Macaulay2()                          # optional - macaulay2
>>> m.set_seed(Integer(123456))                       # optional - macaulay2
123456
>>> [m.random(Integer(100)) for _ in range(Integer(11))]  # optional -␣
↪macaulay2
[8, 29, 5, 22, 4, 32, 35, 57, 3, 95, 36]
```

use(*R*)

Use the Macaulay2 ring R.

EXAMPLES:

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

```
>>> from sage.all import *
>>> # optional - macaulay2
>>> R = macaulay2("QQ[x,y]")
>>> P = macaulay2("ZZ/7[symbol x, symbol y]")
>>> macaulay2("x").cls()._operator('===', P)
true
>>> macaulay2.use(R)
>>> macaulay2("x").cls()._operator('===', R)
true
```

version()

Return the version of Macaulay2 as a tuple.

EXAMPLES:

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

```
>>> from sage.all import *
>>> macaulay2.version() # optional - macaulay2
(1, ...)
```

class sage.interfaces.macaulay2.**Macaulay2Element**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: ExtraTabCompletion, *ExpectElement*, *Macaulay2Element*

Instances of this class represent objects in Macaulay2.

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

sage()

EXAMPLES:

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

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

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

(continues on next page)

(continued from previous page)

```
[3 4]

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

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

sage: # optional - macaulay2
sage: macaulay2(GF(7)).sage()
Finite Field of size 7
sage: macaulay2(GF(49, 'a')).sage()
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: # optional - macaulay2
sage: R = macaulay2("QQ[x,y]")
sage: I = macaulay2("ideal (x,y)")
sage: I.sage()
Ideal (x, y) of Multivariate Polynomial Ring in x, y over Rational Field

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

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

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

(continues on next page)

(continued from previous page)

```
True
sage: macaulay2.substitute('a', R).sage().parent() is R
True

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

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

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

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

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

```
>>> from sage.all import *
>>> # optional - macaulay2
>>> macaulay2(ZZ).sage()         # indirect doctest
Integer Ring
>>> macaulay2(QQ).sage()
Rational Field

>>> # optional - macaulay2
>>> macaulay2(Integer(2)).sage()
2
>>> macaulay2(Integer(1)/Integer(2)).sage()
1/2
>>> macaulay2(Integer(2)/Integer(1)).sage()
2
>>> _.parent()
Rational Field
```

(continues on next page)

(continued from previous page)

```
>>> macaulay2([Integer(1),Integer(2),Integer(3)]).sage()
[1, 2, 3]

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

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

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

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

>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> macaulay2(x**Integer(2)+y**Integer(2)+Integer(1)).sage()      # optional -
↪ macaulay2
x^2 + y^2 + 1

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

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

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

(continues on next page)

(continued from previous page)

```
>>> macaulay2(S).sage() == S
True

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

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

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

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

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

>>> # optional - macaulay2
>>> d = 'digraph ({{1,2},{2,1},{3,1}}, EntryMode => "edges")'
>>> g = macaulay2(d)
>>> g.sage()
Digraph on 3 vertices
>>> g.sage().edges(labels=False)
[(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]
            [a b c]       [ 0  b  a]        [-b]
```

(continues on next page)

(continued from previous page)

```
 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
```

```
>>> from sage.all import *
>>> R = ZZ['a,b,c']
>>> C = macaulay2(ideal(R.gens())).resolution()  # optional - macaulay2
>>> ascii_art(C.sage())                           # optional - macaulay2
                          [-b  0 -c]        [ c]
                          [ a -c  0]        [ a]
             [a b c]      [ 0  b  a]        [-b]
 0 <-- C_0 <-------- C_1 <----------- C_2 <----- C_3 <-- 0
>>> F = C.dot('dd')   # optional - macaulay2
>>> G = F.sage()      # optional - macaulay2
>>> G.in_degree(Integer(2))    # optional - macaulay2
[-b  0 -c]
[ a -c  0]
[ 0  b  a]
>>> F.underscore(Integer(2)).sage() == G.in_degree(Integer(2))  # optional -␣
↪macaulay2
True
>>> (F**Integer(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: # optional - macaulay2
sage: R = macaulay2("ZZ/7[x,y]")
sage: I = macaulay2("ideal (x^3 - y^2)")
sage: (R/I).gens()
{x, y}
sage: (R/I).sage().gens()
(x, y)
```

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

```
>>> from sage.all import *
>>> x, y = (R/I).gens()                        # optional - macaulay2
>>> f = ((x**Integer(3) + Integer(2)*y**Integer(2)*x)**Integer(7)).sage(); f ␣
↪    # optional - macaulay2
2*x*y^18 + y^14
>>> 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 |
sage: B.after_print_text()   # optional - macaulay2
                          2
ZZ-module, submodule of ZZ
```

```
>>> from sage.all import *
>>> B = macaulay2(matrix([[Integer(1), Integer(2)], [Integer(3),␣
↪Integer(6)]])).kernel(); B   # optional - macaulay2
image | 2  |
      | -1 |
>>> 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
```

```
>>> from sage.all import *
>>> macaulay2(ZZ).cls()  # optional - macaulay2
Ring
```

dot(*x*)

EXAMPLES:

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

```
>>> from sage.all import *
>>> # optional - macaulay2
>>> d = macaulay2.new("MutableHashTable")
>>> d["k"] = Integer(4)
>>> d.dot("k")
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])'
```

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

0      1      2      3
```

```
>>> from sage.all import *
>>> # optional - macaulay2
>>> S = macaulay2(QQ['x,y'])
>>> S.name()
'sage...'
>>> R = S.name("R")
>>> R.name()
'R'
>>> R.vars().cokernel().resolution()
 1      2      1
R  <-- R  <-- R  <-- 0
<BLANKLINE>
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
```

```
>>> from sage.all import *
>>> A = macaulay2(ZZ['x,y,z'], name='A')  # optional - macaulay2
>>> A.name()                              # optional - macaulay2
'A'
>>> A**Integer(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: # optional - macaulay2
sage: R = macaulay2.ring('QQ','(x,y)')
```

(continues on next page)

(continued from previous page)

```
sage: f = macaulay2('x^3 + 3*y^11 + 5')
sage: print(f)
 3     11
x  + 3y   + 5
sage: f.sage_polystring()
'x**3+3*y**11+5'
```

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

sharp(*x*)

EXAMPLES:

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

```
>>> from sage.all import *
>>> a = macaulay2([Integer(1),Integer(2),Integer(3)]) # optional - macaulay2
>>> a.sharp(Integer(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)}
```

```
>>> from sage.all import *
>>> a = macaulay2([Integer(1),Integer(2)]).set()   # optional - macaulay2
>>> a.starstar(a)                # optional - macaulay2
set {(1, 1), (1, 2), (2, 1), (2, 2)}
```

structure_sheaf()

EXAMPLES:

```
sage: # optional - macaulay2
sage: S = macaulay2('QQ[a..d]')
sage: R = S / macaulay2('a^3 + b^3 + c^3 + d^3')
sage: X = R.Proj().name('X')
sage: X.structure_sheaf()
doctest:...: DeprecationWarning: The function `structure_sheaf` is deprecated.
```

(continues on next page)

(continued from previous page)

```
↪ Use `self.sheaf()` instead.
See https://github.com/sagemath/sage/issues/27848 for details.
OO
  X
sage: X.sheaf()
OO
  X
```

```
>>> from sage.all import *
>>> # optional - macaulay2
>>> S = macaulay2('QQ[a..d]')
>>> R = S / macaulay2('a^3 + b^3 + c^3 + d^3')
>>> X = R.Proj().name('X')
>>> X.structure_sheaf()
doctest:...: DeprecationWarning: The function `structure_sheaf` is deprecated.
↪ Use `self.sheaf()` instead.
See https://github.com/sagemath/sage/issues/27848 for details.
OO
  X
>>> X.sheaf()
OO
  X
```

subs(**args*, ***kwds*)

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

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

to_sage(**args*, ***kwds*)

Deprecated: Use `sage()` instead. See Issue #27848 for details.

underscore(*x*)

EXAMPLES:

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

```
>>> from sage.all import *
>>> a = macaulay2([Integer(1),Integer(2),Integer(3)])  # optional - macaulay2
>>> a.underscore(Integer(0))         # optional - macaulay2
1
```

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

Bases: *ExpectFunction*

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

Bases: *FunctionElement*

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

Return `True` if `x` is a *Macaulay2Element*.

This function is deprecated; use `isinstance()` (of *sage.interfaces.abc.Macaulay2Element*) instead.

EXAMPLES:

```
sage: from sage.interfaces.macaulay2 import is_Macaulay2Element
sage: is_Macaulay2Element(2)              # optional - macaulay2
doctest:...: DeprecationWarning: the function is_Macaulay2Element is deprecated;␣
↪use isinstance(x, sage.interfaces.abc.Macaulay2Element) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
sage: is_Macaulay2Element(macaulay2(2))   # optional - macaulay2
True
```

```
>>> from sage.all import *
>>> from sage.interfaces.macaulay2 import is_Macaulay2Element
>>> is_Macaulay2Element(Integer(2))              # optional - macaulay2
doctest:...: DeprecationWarning: the function is_Macaulay2Element is deprecated;␣
↪use isinstance(x, sage.interfaces.abc.Macaulay2Element) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
>>> is_Macaulay2Element(macaulay2(Integer(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
...
```

```
>>> from sage.all import *
>>> macaulay2_console()                    # not tested
```

(continues on next page)

(continued from previous page)

```
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
```

```
>>> from sage.all import *
>>> from sage.interfaces.macaulay2 import reduce_load_macaulay2
>>> 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'
```

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

CHAPTER

TWENTYTHREE

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 https://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`.

23.1 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
```

```
>>> from sage.all import *
>>> E = magma('EllipticCurve([0,1,1,-1,0])')                    # optional - magma
>>> E.Rank(Bound = Integer(5))                                            # optional -
↪magma
0
```

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

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

We verify that an obviously principal ideal is principal:

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

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

23.3 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
```

```
>>> from sage.all import *
>>> t = '"%s"'%Integer(10)**Integer(10000)   # ten thousand character string.       #␣
↪optional - magma
>>> a = magma.eval(t)                                             # optional - magma
>>> a = magma(t)                                                  # optional - magma
```

23.4 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
```

```
>>> from sage.all import *
>>> m=matrix(ZZ,Integer(2),[Integer(1),Integer(2),Integer(3),Integer(4)])             ␣
↪                          # optional - magma
```

Then I create a corresponding matrix A in Magma:

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

```
>>> from sage.all import *
>>> A = magma(m)                                                  # optional - magma
```

It is called _sage_[...] in Magma:

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

```
>>> from sage.all import *
>>> s = A.name(); s                                               # optional - magma
'_sage_[...]'
```

It's there:

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

```
>>> from sage.all import *
>>> magma.eval(s)                                                 # optional - magma
'[1 2]\n[3 4]'
```

Now I delete the reference to that matrix:

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

```
>>> from sage.all import *
>>> del A                                                # optional - magma
```

Now _sage_[...] is "zeroed out" in the Magma session:

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

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

23.5 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 + ...
]
```

```
>>> from sage.all import *
>>> N = Integer(20)
>>> D = Integer(20)
>>> eps_top = fundamental_discriminant(D)
>>> eps = magma.KroneckerCharacter(eps_top, RationalField())        # optional - magma
>>> M2 = magma.ModularForms(eps)                                    # optional - magma
>>> print(M2)                                                       # optional - magma
Space of modular forms on Gamma_1(5) ...
>>> 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: # optional - magma
sage: G = magma.DirichletGroup(20)
sage: G.AssignNames(['a', 'b'])
sage: (G.1).Modulus()
20
sage: e = magma.DirichletGroup(40)(G.1)
sage: print(e)
Kronecker character -4 in modulus 40
sage: print(e.Modulus())
40
```

```
>>> from sage.all import *
>>> # optional - magma
>>> G = magma.DirichletGroup(Integer(20))
>>> G.AssignNames(['a', 'b'])
>>> (G.gen(1)).Modulus()
20
>>> e = magma.DirichletGroup(Integer(40))(G.gen(1))
>>> print(e)
Kronecker character -4 in modulus 40
>>> print(e.Modulus())
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
sage: S.1                                                             # optional -␣
↪magma
y
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=('y',)); (y,) = R._first_ngens(1)
>>> S = magma(R)                                                  # optional - magma
>>> print(S)                                                      # optional - magma
Univariate Polynomial Ring in y over Rational Field
>>> S.gen(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)
```

(continues on next page)

(continued from previous page)

```
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
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> x * Integer(5)
5*x
>>> x * RealNumber('1.0')
x
>>> x * (Integer(2)/Integer(3))
2/3*x
>>> y = magma(x)                                                # optional - magma
>>> y * Integer(5)                                                       #␣
↪optional - magma
5*x
>>> y * RealNumber('1.0')                                                    #␣
↪optional - magma
$.1
>>> y * (Integer(2)/Integer(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_tmpdir=None*, *user_config=False*, *seed=None*, *command=None*)

Bases: `ExtraTabCompletion`, *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.SetDefaultRealFieldPrecision(200, nvals=0)  # magma >= v2.12;␣
↪optional - magma
sage: magma.eval('1.1')   # optional - magma
'1.1000000000000000000000000000000000000000000000000000000000000000000000000000000
↪000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪0000000000000000000000000000000000000000'
sage: magma.SetDefaultRealFieldPrecision(30, nvals=0)  # optional - magma
```

```
>>> from sage.all import *
>>> magma.SetDefaultRealFieldPrecision(Integer(200), nvals=Integer(0))  # magma >
↪= v2.12; optional - magma
>>> magma.eval('1.1')   # optional - magma
'1.1000000000000000000000000000000000000000000000000000000000000000000000000000000
↪000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪0000000000000000000000000000000000000000'
>>> magma.SetDefaultRealFieldPrecision(Integer(30), nvals=Integer(0))  # optional␣
↪- magma
```

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` – 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
```

```
>>> from sage.all import *
>>> SAGE_EXTCODE = SAGE_ENV['SAGE_EXTCODE']                # optional - magma
>>> 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...
```

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

AttachSpec(*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` – 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)
```

```
>>> from sage.all import *
>>> SAGE_EXTCODE = SAGE_ENV['SAGE_EXTCODE']             # optional - magma
>>> magma.attach_spec('%s/magma/spec'%SAGE_EXTCODE)     # optional - magma
>>> 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)
```

GetNthreads()

Get the number of threads used in Magma.

EXAMPLES:

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

```
>>> from sage.all import *
>>> magma.set_nthreads(Integer(2))                    #optional - magma
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

```
>>> from sage.all import *
>>> magma.set_verbose("Groebner", Integer(2))         # optional - magma
>>> 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
```

```
>>> from sage.all import *
>>> magma.set_nthreads(Integer(2))                     #optional - magma
>>> 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
```

```
>>> from sage.all import *
>>> magma.set_verbose("Groebner", Integer(2))      # optional - magma
>>> 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` – 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
```

```
>>> from sage.all import *
>>> SAGE_EXTCODE = SAGE_ENV['SAGE_EXTCODE']                    # optional - magma
>>> 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...
```

```
>>> from sage.all import *
>>> SAGE_EXTCODE = SAGE_ENV['SAGE_EXTCODE']                      # optional - magma
>>> 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` – 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
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
RuntimeError: Can't open package spec file .../magma/spec2 for reading (No␣
↪such file or directory)
```

```
>>> from sage.all import *
>>> SAGE_EXTCODE = SAGE_ENV['SAGE_EXTCODE']            # optional - magma
>>> magma.attach_spec('%s/magma/spec'%SAGE_EXTCODE)     # optional - magma
>>> 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: # optional - magma
sage: V = magma.RModule(ZZ,3)
sage: V
RModule(IntegerRing(), 3)
sage: magma.bar_call(V, 'quo', [[1,2,3]], nvals=1)
RModule(IntegerRing(), 2)
sage: magma.bar_call(V, 'quo', [[1,2,3]], nvals=2)
(RModule(IntegerRing(), 2),
 Mapping from: RModule(IntegerRing(), 3) to RModule(IntegerRing(), 2))
sage: magma.bar_call(V, 'quo', V, nvals=2)
(RModule(IntegerRing(), 0),
 Mapping from: RModule(IntegerRing(), 3) to RModule(IntegerRing(), 0))
```

```
>>> from sage.all import *
>>> # optional - magma
>>> V = magma.RModule(ZZ,Integer(3))
>>> V
RModule(IntegerRing(), 3)
>>> magma.bar_call(V, 'quo', [[Integer(1),Integer(2),Integer(3)]],␣
↪nvals=Integer(1))
```

(continues on next page)

(continued from previous page)

```
RModule(IntegerRing(), 2)
>>> magma.bar_call(V, 'quo', [[Integer(1),Integer(2),Integer(3)]],␣
↪nvals=Integer(2))
(RModule(IntegerRing(), 2),
 Mapping from: RModule(IntegerRing(), 3) to RModule(IntegerRing(), 2))
>>> magma.bar_call(V, 'quo', V, nvals=Integer(2))
(RModule(IntegerRing(), 0),
 Mapping from: RModule(IntegerRing(), 3) to RModule(IntegerRing(), 0))
```

chdir(*dir*)

Change the Magma interpreter's current working directory.

INPUT:

- `dir` – string

EXAMPLES:

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

```
>>> from sage.all import *
>>> magma.chdir('/')                  # optional - magma
>>> magma.eval('System("pwd")')       # optional - magma
'/'
```

clear(*var*)

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

INPUT:

- `var` – 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'
```

```
>>> from sage.all import *
>>> magma = Magma()       # optional - magma
>>> magma.clear('foo')    # sets foo to 0 in magma; optional - magma
>>> 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'
```

```
>>> from sage.all import *
>>> a = magma('10')         # optional - magma
>>> a.name()                # optional - magma
'foo'
```

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

```
sage: # optional - magma
sage: magma = Magma()
sage: a = magma('100')
sage: a.name()
'_sage_[1]'
sage: del a
sage: b = magma('257')
sage: b.name()
'_sage_[1]'
sage: del b
sage: magma('_sage_[1]')
0
```

```
>>> from sage.all import *
>>> # optional - magma
>>> magma = Magma()
>>> a = magma('100')
>>> a.name()
'_sage_[1]'
>>> del a
>>> b = magma('257')
>>> b.name()
'_sage_[1]'
>>> del b
>>> magma('_sage_[1]')
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
```

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

```
>>> from sage.all import *
>>> # optional - magma
>>> type(magma.cputime())
<... 'float'>
>>> magma.cputime()   # random
1.9399999999999999
>>> t = magma.cputime()
>>> magma.cputime(t)   # random
0.02
```

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

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'
```

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

```
>>> from sage.all import *
>>> magma.eval("a := %s;"%(Integer(10)**Integer(10000)))    # optional - magma
''
```

Verify that Issue #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'
```

```
>>> from sage.all import *
>>> nl=chr(Integer(10)) # newline character
>>> 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 Issue #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 ]'
```

```
>>> from sage.all import *
>>> nl=chr(Integer(10)) # newline character
>>> magma.eval("a:=3;"+nl+"b:=5;") == nl  # optional - magma
True
>>> 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: instance of `MagmaElement` or a tuple of `nvals` many `MagmaElement` instances

EXAMPLES:

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

```
>>> from sage.all import *
>>> magma.function_call('Factorization', Integer(100))    # optional - magma
[ <2, 2>, <5, 2> ]
>>> magma.function_call('NextPrime', Integer(100), {'Proof':False})    #␣
↪optional - magma
101
>>> magma.function_call('PolynomialRing', [QQ,Integer(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)
```

```
>>> from sage.all import *
>>> magma.function_call('IsSquare', Integer(100))         # optional - magma
true
>>> magma.function_call('IsSquare', Integer(100), nvals=Integer(2))     #␣
↪optional - magma
(true, 10)
>>> magma.function_call('IsSquare', Integer(100), nvals=Integer(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 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'
```

```
>>> from sage.all import *
>>> magma.set('abc', '2 + 3/5')      # optional - magma
>>> 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
```

```
>>> from sage.all import *
>>> magma.set_nthreads(Integer(2))                #optional - magma
>>> 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_verbose("Groebner", 2)        # optional - magma
sage: magma.get_verbose("Groebner")           # optional - magma
2
```

```
>>> from sage.all import *
>>> magma.set_verbose("Groebner", Integer(2))        # optional - magma
>>> magma.get_verbose("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
...
```

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

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> magma.ideal([x**Integer(2), y**Integer(3)*x])          # optional - magma
Ideal of Polynomial ring of rank 2 over Rational Field
Order: Graded Reverse Lexicographical
Variables: x, y
Homogeneous
Basis:
```

(continues on next page)

(continued from previous page)

```
[
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 will 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 "....m"
hi
sage: magma('f(12)')  # optional - magma
144
```

```
>>> from sage.all import *
>>> from tempfile import NamedTemporaryFile as NTF
>>> 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 "....m"
hi
>>> 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)
```

```
>>> from sage.all import *
>>> R = magma.objgens('PolynomialRing(Rationals(),2)', 'alpha,beta')     #␣
↪optional - magma
>>> 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
Order: Lexicographical
Variables: X, Y
sage: S                                    # optional - magma
Polynomial ring of rank 2 over Rational Field
Order: Lexicographical
Variables: X, Y
```

```
>>> from sage.all import *
>>> S = magma.objgens(R, 'X,Y')          # optional - magma
>>> R                                    # optional - magma
Polynomial ring of rank 2 over Rational Field
Order: Lexicographical
Variables: X, Y
>>> 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
```

```
>>> from sage.all import *
>>> magma.set('abc', '2 + 3/5')        # optional - magma
>>> 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
```

```
>>> from sage.all import *
>>> magma.set_nthreads(Integer(2))                    #optional - magma
>>> 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]
```

```
>>> from sage.all import *
>>> m = Magma() # optional - magma
>>> m.set_seed(Integer(1)) # optional - magma
1
>>> [m.Random(Integer(100)) for i in range(Integer(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
```

```
>>> from sage.all import *
>>> magma.set_verbose("Groebner", Integer(2))      # optional - magma
>>> 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')
```

```
>>> from sage.all import *
>>> magma.version()        # random, optional - magma
((2, 14, 9), 'V2.14-9')
```

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

Bases: `ExtraTabCompletion`, *`ExpectElement`*, *`MagmaElement`*

`AssignNames`(*names*)

EXAMPLES:

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

```
>>> from sage.all import *
>>> # optional - magma
>>> S = magma.PolynomialRing(magma.Integers(), Integer(2))
>>> S.AssignNames(['a', 'b'])
>>> S.gen(1)
a
>>> S.gen(1)**Integer(2) + S.gen(2)
a^2 + b
```

`assign_names`(*names*)

EXAMPLES:

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

```
>>> from sage.all import *
>>> # optional - magma
>>> S = magma.PolynomialRing(magma.Integers(), Integer(2))
>>> S.AssignNames(['a', 'b'])
>>> S.gen(1)
a
>>> S.gen(1)**Integer(2) + S.gen(2)
a^2 + b
```

eval(**args*)

Evaluate `self` at the inputs.

INPUT:

- `*args` – import arguments

OUTPUT: self(*args)

EXAMPLES:

```
sage: # optional - magma
sage: f = magma('Factorization')
sage: f.evaluate(15)
[ <3, 1>, <5, 1> ]
sage: f(15)
[ <3, 1>, <5, 1> ]
sage: f = magma('GCD')
sage: f.evaluate(15,20)
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 ]
```

```
>>> from sage.all import *
>>> # optional - magma
>>> f = magma('Factorization')
>>> f.evaluate(Integer(15))
[ <3, 1>, <5, 1> ]
>>> f(Integer(15))
[ <3, 1>, <5, 1> ]
>>> f = magma('GCD')
>>> f.evaluate(Integer(15),Integer(20))
5

>>> m = matrix(QQ, Integer(2), Integer(2), [Integer(2),Integer(3),Integer(5),
↪Integer(7)])       # optional - magma
>>> f = magma('ElementaryDivisors')       # optional - magma
>>> f.evaluate(m)                         # optional - magma
[ 1, 1 ]
```

evaluate(**args*)

Evaluate `self` at the inputs.

INPUT:

- `*args` – import arguments

OUTPUT: self(*args)

EXAMPLES:

```
sage: # optional - magma
sage: f = magma('Factorization')
sage: f.evaluate(15)
```

(continues on next page)

(continued from previous page)

```
[ <3, 1>, <5, 1> ]
sage: f(15)
[ <3, 1>, <5, 1> ]
sage: f = magma('GCD')
sage: f.evaluate(15,20)
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 ]
```

```
>>> from sage.all import *
>>> # optional - magma
>>> f = magma('Factorization')
>>> f.evaluate(Integer(15))
[ <3, 1>, <5, 1> ]
>>> f(Integer(15))
[ <3, 1>, <5, 1> ]
>>> f = magma('GCD')
>>> f.evaluate(Integer(15),Integer(20))
5

>>> m = matrix(QQ, Integer(2), Integer(2), [Integer(2),Integer(3),Integer(5),
↪Integer(7)])       # optional - magma
>>> f = magma('ElementaryDivisors')       # optional - magma
>>> 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 – *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
w
sage: m.gen(0)                 # optional -- magma
```

(continues on next page)

(continued from previous page)

```
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: tuple index out of range
```

```
>>> from sage.all import *
>>> k = GF(Integer(9), names=('a',)); (a,) = k._first_ngens(1)
>>> magma(k).gen(Integer(1))         # optional -- magma
a
>>> R = k['s, t, w']; (s, t, w,) = R._first_ngens(3)
>>> m = magma(R)             # optional -- magma
>>> m.gen(Integer(1))                # optional -- magma
s
>>> m.gen(Integer(2))                # optional -- magma
t
>>> m.gen(Integer(3))                # optional -- magma
w
>>> m.gen(Integer(0))                # optional -- magma
Traceback (most recent call last):
...
IndexError: index must be positive since Magma indexes are 1-based
>>> m.gen(Integer(4))                # optional -- magma
Traceback (most recent call last):
...
IndexError: tuple 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
```

```
>>> from sage.all import *
>>> R = QQ['x, zw']; (x, zw,) = R._first_ngens(2)
>>> S = magma(R)                 # optional - magma
>>> S.gen_names()                # optional - magma
```

(continues on next page)

(continued from previous page)

```
('_sage_[...]', '_sage_[...]')
>>> magma(S.gen_names()[Integer(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 tuple of the resulting X.i. Note - I do not 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,)
```

```
>>> from sage.all import *
>>> magma("VectorSpace(RationalField(),3)").gens()         # optional - magma
((1 0 0), (0 1 0), (0 0 1))
>>> magma("AbelianGroup(EllipticCurve([1..5]))").gens()    # optional - magma
($.1,)
```

get_magma_attribute(*attrname*)

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

OUTPUT: *MagmaElement*

EXAMPLES:

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

```
>>> from sage.all import *
>>> # optional - magma
>>> V = magma("VectorSpace(RationalField(),10)")
>>> V.set_magma_attribute('M','"hello"')
>>> V.get_magma_attribute('M')
hello
>>> V.M
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: # optional - magma
sage: R = magma('PolynomialRing(RationalField())')
sage: R.assign_names(['x'])
sage: x = R.1
sage: R.ideal([x^2 - 1, x^3 - 1])
Ideal of Univariate Polynomial Ring in x over Rational Field generated by x -␣
↪1
```

```
>>> from sage.all import *
>>> # optional - magma
>>> R = magma('PolynomialRing(RationalField())')
>>> R.assign_names(['x'])
>>> x = R.gen(1)
>>> R.ideal([x**Integer(2) - Integer(1), x**Integer(3) - Integer(1)])
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']
```

```
>>> from sage.all import *
>>> V = magma("VectorSpace(RationalField(),2)")        # optional - magma
>>> v = V.list_attributes(); v.sort()                 # optional - magma
>>> 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` – boolean (default: `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
"'*'..."
```

```
>>> from sage.all import *
>>> v = magma('2/3').methods()              # optional - magma
>>> v[Integer(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)
```

```
>>> from sage.all import *
>>> V = magma('VectorSpace(RationalField(),3)')       # optional - magma
>>> V.quo([[Integer(1),Integer(2),Integer(3)], [Integer(1),Integer(1),
↪Integer(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)
```

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

```
>>> from sage.all import *
>>> # optional - magma
>>> V = magma.RModule(ZZ,Integer(3)); V
RModule(IntegerRing(), 3)
>>> W, phi = V.quo([[Integer(1),Integer(2),Integer(3)]])
>>> W
RModule(IntegerRing(), 2)
>>> phi
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: # optional - magma
sage: V = magma("VectorSpace(RationalField(),2)")
sage: V.set_magma_attribute('M',10)
sage: V.get_magma_attribute('M')
10
sage: V.M
10
```

```
>>> from sage.all import *
>>> # optional - magma
>>> V = magma("VectorSpace(RationalField(),2)")
>>> V.set_magma_attribute('M',Integer(10))
>>> V.get_magma_attribute('M')
10
>>> V.M
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)
Echelonized basis:
(1 0 1)
(0 1 1)
```

```
>>> from sage.all import *
>>> V = magma('VectorSpace(RationalField(),3)')       # optional - magma
>>> W = V.sub([ [Integer(1),Integer(2),Integer(3)], [Integer(1),Integer(1),
↪Integer(2)] ]); W                # optional - magma
Vector space of degree 3, dimension 2 over Rational Field
Generators:
(1 2 3)
(1 1 2)
Echelonized basis:
(1 0 1)
(0 1 1)
```

class sage.interfaces.magma.**MagmaFunction**(*parent*, *name*)

Bases: *ExpectFunction*

class sage.interfaces.magma.**MagmaFunctionElement**(*obj*, *name*)

Bases: *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.

app_inpt = re.compile('^Append\\(~_sage_, 0\\);$')

cmd_inpt = re.compile('^>>>$')

deg_curr = re.compile('^Basis length\\: (\\d+), queue length\\: (\\d+), step degree\\: (\\d+), num pairs\\: (\\d+)$')

flush()

EXAMPLES:

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

```
>>> from sage.all import *
>>> from sage.interfaces.magma import MagmaGBLogPrettyPrinter
>>> logs = MagmaGBLogPrettyPrinter()
>>> logs.flush()
```

pol_curr = re.compile('^Number of pair polynomials\\: (\\d+), at (\\d+) column\\(s\\), .*')

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: ...
```

```
>>> from sage.all import *
>>> P = GF(Integer(32003))['x, y, z']; (x, y, z,) = P._first_ngens(3)
>>> I = sage.rings.ideal.Katsura(P)
>>> _ = 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/'
```

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

EXAMPLES:

```
sage: from sage.interfaces.magma import is_MagmaElement
sage: is_MagmaElement(2)
```

(continues on next page)

(continued from previous page)

```
doctest:...: DeprecationWarning: the function is_MagmaElement is deprecated; use
↪isinstance(x, sage.interfaces.abc.MagmaElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
sage: is_MagmaElement(magma(2))                    # optional - magma
True
```

```
>>> from sage.all import *
>>> from sage.interfaces.magma import is_MagmaElement
>>> is_MagmaElement(Integer(2))
doctest:...: DeprecationWarning: the function is_MagmaElement is deprecated; use
↪isinstance(x, sage.interfaces.abc.MagmaElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
>>> is_MagmaElement(magma(Integer(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
```

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

```
>>> from sage.all import *
>>> P = PolynomialRing(GF(Integer(127)), names=('a', 'b', 'c', 'd', 'e',)); (a, b,
↪ c, d, e,) = P._first_ngens(5)
>>> J = sage.rings.ideal.Cyclic(P).homogenize()
>>> from sage.misc.sageinspect import sage_getsource
>>> "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
```

```
>>> from sage.all import *
>>> sage.interfaces.magma.reduce_load_Magma()
Magma
```

CHAPTER

TWENTYFOUR

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> ]
```

```
>>> from sage.all import *
>>> magma_free("Factorization(9290348092384)")  # optional - internet
[ <2, 5>, <290323377887, 1> ]
```

eval(*x*, ***kwds*)

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.

Warning

The code must evaluate in at most 120 seconds and there is a limitation on the amount of RAM.

EXAMPLES:

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

```
>>> from sage.all import *
>>> magma_free("Factorization(9290348092384)")  # optional - internet
[ <2, 5>, <290323377887, 1> ]
```

CHAPTER

TWENTYFIVE

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

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

25.1 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)
   ```

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

2. 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)
   ```

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

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

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

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

```
>>> from sage.all import *
>>> maple.console()          # not tested
>>> !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
```

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

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

```
>>> from sage.all import *
>>> maple.fibonacci(Integer(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
```

```
>>> from sage.all import *
>>> maple('combinat[fibonacci]')(Integer(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
```

```
>>> from sage.all import *
>>> maple.with_package('combinat')        # optional - maple
```

or

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

```
>>> from sage.all import *
>>> maple.load('combinat')                # optional - maple
```

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

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

```
>>> from sage.all import *
>>> maple.fibonacci(Integer(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
```

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

```
>>> from sage.all import *
>>> maple.isprime(maple.fibonacci(Integer(27)))      # optional - maple
false
>>> maple.ifactor(maple.fibonacci(Integer(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: # optional - maple
sage: alpha = maple('(1+sqrt(5))/2')
sage: beta = maple('(1-sqrt(5))/2')
sage: f19  = alpha^19 - beta^19/maple('sqrt(5)')
sage: f19
(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
6765+5778/5*5^(1/2)
```

```
>>> from sage.all import *
>>> # optional - maple
>>> alpha = maple('(1+sqrt(5))/2')
>>> beta = maple('(1-sqrt(5))/2')
>>> f19  = alpha**Integer(19) - beta**Integer(19)/maple('sqrt(5)')
>>> f19
(1/2+1/2*5^(1/2))^19-1/5*(1/2-1/2*5^(1/2))^19*5^(1/2)
>>> f19.simplify()          # somewhat randomly ordered output
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
```

```
>>> from sage.all import *
>>> mysqcu = maple('proc(x) if x > 0 then x^2 else x^3 fi end')     # optional - maple
>>> mysqcu(Integer(5))                                                         #␣
↪optional - maple
25
>>> mysqcu(-Integer(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_tmpdir=None*, *logfile=None*, *ulimit=None*)

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

```
>>> from sage.all import *
>>> # optional - maple
>>> maple.set('xx', '2')
>>> maple.get('xx')
'2'
>>> maple.clear('xx')
>>> maple.get('xx')
'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
```

```
>>> from sage.all import *
>>> c = maple.completions('di')  # optional - maple
>>> 'divide' in c                # optional - maple
True
```

console()

Spawn a new Maple command-line session.

EXAMPLES:

```
sage: maple.console()  # not tested
    |\^/|     Maple 2019 (X86 64 LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2019
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
```

```
>>> from sage.all import *
>>> maple.console()  # not tested
    |\^/|     Maple 2019 (X86 64 LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2019
 \  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: # optional - maple
sage: t = maple.cputime()
sage: t                    # random
0.02
sage: x = maple('x')
sage: maple.diff(x^2, x)
2*x
sage: maple.cputime(t)     # random
0.0
```

```
>>> from sage.all import *
>>> # optional - maple
>>> t = maple.cputime()
>>> t                    # random
0.02
>>> x = maple('x')
>>> maple.diff(x**Integer(2), x)
2*x
>>> maple.cputime(t)     # random
0.0
```

expect ()

Return the pexpect object for this Maple session.

EXAMPLES:

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

```
>>> from sage.all import *
>>> # optional - maple
>>> m = Maple()
>>> m.expect() is None
```

(continues on next page)

(continued from previous page)

```
True
>>> m._start()
>>> m.expect()
Maple with PID ...
>>> m.quit()
```

get(*var*)

Get the value of the variable `var`.

EXAMPLES:

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

```
>>> from sage.all import *
>>> maple.set('xx', '2') # optional - maple
>>> 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` – string to search for in the maple help system

EXAMPLES:

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

```
>>> from sage.all import *
>>> 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: # optional - maple
sage: maple.quit()   # reset maple
sage: maple('partition(10)')
partition(10)
sage: maple('bell(10)')
bell(10)
```

(continues on next page)

(continued from previous page)

```
sage: maple.with_package('combinat')
sage: maple('partition(10)')
[[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)')
115975
sage: maple('fibonacci(10)')
55
```

```
>>> from sage.all import *
>>> # optional - maple
>>> maple.quit()    # reset maple
>>> maple('partition(10)')
partition(10)
>>> maple('bell(10)')
bell(10)
>>> maple.with_package('combinat')
>>> maple('partition(10)')
[[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]]
>>> maple('bell(10)')
115975
>>> maple('fibonacci(10)')
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'
```

```
>>> from sage.all import *
>>> maple.set('xx', '2') # optional - maple
>>> 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 – string representing the function whose source code you want

EXAMPLES:

```
sage: maple.source('curry')  # not tested
... -> subs('_X' = _passed[2 .. _npassed],() -> ...(_X, _passed))
```

```
>>> from sage.all import *
>>> maple.source('curry')  # not tested
... -> subs('_X' = _passed[2 .. _npassed],() -> ...(_X, _passed))
```

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: # optional - maple
sage: maple.quit()   # reset maple
sage: maple('partition(10)')
partition(10)
sage: maple('bell(10)')
bell(10)
sage: maple.with_package('combinat')
sage: maple('partition(10)')
[[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)')
115975
sage: maple('fibonacci(10)')
55
```

```
>>> from sage.all import *
>>> # optional - maple
>>> maple.quit()   # reset maple
>>> maple('partition(10)')
partition(10)
>>> maple('bell(10)')
```

(continues on next page)

(continued from previous page)

```
bell(10)
>>> maple.with_package('combinat')
>>> maple('partition(10)')
[[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]]
>>> maple('bell(10)')
115975
>>> maple('fibonacci(10)')
55
```

class sage.interfaces.maple.**MapleElement**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: ExtraTabCompletion, *ExpectElement*

op(*i=None*)

Return the i-th operand of this expression.

INPUT:

- i – 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}
```

```
>>> from sage.all import *
>>> V = maple(vector(QQ,[Integer(4),Integer(5),Integer(6)]))       # optional -
↪ maple
>>> V.op(Integer(1))                             # optional - maple
3
>>> V.op(Integer(2))                             # optional - maple
{1 = 4, 2 = 5, 3 = 6}
```

class sage.interfaces.maple.**MapleFunction**(*parent*, *name*)

Bases: *ExpectFunction*

class sage.interfaces.maple.**MapleFunctionElement**(*obj*, *name*)

Bases: *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.
>
```

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

```
>>> from sage.all import *
>>> from sage.interfaces.maple import reduce_load_Maple
>>> reduce_load_Maple()
Maple
```

CHAPTER

TWENTYSIX

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())
...
```

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

```
>>> from sage.all import *
>>> mobj = mathematica(x**Integer(2)-Integer(1))             # optional - mathematica
>>> 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: # optional - mathematica
sage: mobj = mathematica(x^2-1)
sage: mobj2 = mobj.Factor(); mobj2
(-1 + x)*(1 + x)
sage: mobj2.parent()
Mathematica
sage: sobj = mobj2.sage(); sobj
(x + 1)*(x - 1)
```

(continues on next page)

(continued from previous page)

```
sage: sobj.parent()
Symbolic Ring
```

```
>>> from sage.all import *
>>> # optional - mathematica
>>> mobj = mathematica(x**Integer(2)-Integer(1))
>>> mobj2 = mobj.Factor(); mobj2
(-1 + x)*(1 + x)
>>> mobj2.parent()
Mathematica
>>> sobj = mobj2.sage(); sobj
(x + 1)*(x - 1)
>>> sobj.parent()
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}
```

```
>>> from sage.all import *
>>> mathematica('Factor[x^2-1]')          # optional - mathematica
(-1 + x)*(1 + x)
>>> 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]'
```

```
>>> from sage.all import *
>>> mathematica.eval('x^2 - 1')           # optional - mathematica
               2
         -1 + x
>>> repr(mathematica('Range[3]'))         # optional - mathematica
'{1, 2, 3}'
>>> 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!)

26.1 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.

26.1.1 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
```

```
>>> from sage.all import *
>>> m = mathematica
>>> a = m(Integer(1)) + m(Integer(1)); a                # optional - mathematica
2
>>> a.parent()                        # optional - mathematica
Mathematica
>>> m('1+1')                          # optional - mathematica
2
>>> m(Integer(3))**m(Integer(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
```

```
>>> from sage.all import *
>>> m = mathematica
>>> m(Integer(2)).Plus(m(Integer(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
```

```
>>> from sage.all import *
>>> m(Integer(7)).Times(m(Integer(2)).Plus(m(Integer(3))))        # optional -␣
↪mathematica
35
>>> m('7(2+3)')                          # optional - mathematica
35
```

26.1.2 Some typical input

We solve an equation and a system of two equations:

```
sage: # optional - mathematica
sage: eqn = mathematica('3x + 5 == 14')
sage: eqn
5 + 3*x == 14
sage: eqn.Solve('x')
{{x -> 3}}
sage: sys = mathematica('{x^2 - 3y == 3, 2x - y == 1}')
sage: print(sys)
           2
         {x  - 3 y == 3, 2 x - y == 1}
sage: sys.Solve('{x, y}')
{{x -> 0, y -> -1}, {x -> 6, y -> 11}}
```

```
>>> from sage.all import *
>>> # optional - mathematica
>>> eqn = mathematica('3x + 5 == 14')
>>> eqn
5 + 3*x == 14
>>> eqn.Solve('x')
{{x -> 3}}
>>> sys = mathematica('{x^2 - 3y == 3, 2x - y == 1}')
>>> print(sys)
           2
         {x  - 3 y == 3, 2 x - y == 1}
>>> sys.Solve('{x, y}')
{{x -> 0, y -> -1}, {x -> 6, y -> 11}}
```

26.1.3 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
```

```
>>> from sage.all import *
>>> c = m(Integer(5))                           # optional - mathematica
>>> print(m('b + c x'))                  # optional - mathematica
             b + c x
```

(continues on next page)

(continued from previous page)

```
>>> 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}}
```

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

26.1.4 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
```

```
>>> from sage.all import *
>>> m = mathematica
>>> _ = mathematica('f[p_] = p^2');    # optional - mathematica
>>> m('f[9]')                         # optional - mathematica
81
```

26.1.5 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}
```

```
>>> from sage.all import *
>>> eqn = mathematica('Exp[x] - 3x == 0') # optional - mathematica
>>> eqn.FindRoot(['x', Integer(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)')
1.5121345516578424738967396780720387046
```

```
>>> from sage.all import *
>>> gp('solve(x=1,2,exp(x)-3*x)')
1.5121345516578424738967396780720387046
```

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}}
```

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

```
>>> from sage.all import *
>>> R = PolynomialRing(ZZ, names=('x',)); (x,) = R._first_ngens(1)
>>> f = (x**Integer(100)+Integer(17)*x+Integer(5))*(x**Integer(100)-
↪Integer(5)*x+Integer(20))
>>> f
x^200 + 12*x^101 + 25*x^100 - 85*x^2 + 315*x + 100
>>> g = mathematica(str(f))            # optional - mathematica
>>> print(g)                           # optional - mathematica
                           2        100        101    200
         100 + 315 x - 85 x  + 25 x    + 12 x    + x
>>> g                                  # optional - mathematica
```

(continues on next page)

(continued from previous page)

```
100 + 315*x - 85*x^2 + 25*x^100 + 12*x^101 + x^200
>>> 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 )
```

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

We factor an integer:

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

```
>>> from sage.all import *
>>> # optional - mathematica
>>> n = mathematica(Integer(2434500))
>>> n.FactorInteger()
{{2, 2}, {3, 2}, {5, 3}, {541, 1}}
>>> n = mathematica(Integer(2434500))
>>> F = n.FactorInteger(); F
{{2, 2}, {3, 2}, {5, 3}, {541, 1}}
>>> F[Integer(1)]
{2, 2}
>>> F[Integer(4)]
{541, 1}
```

Mathematica's ECM package is no longer available.

26.2 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
```

```
>>> from sage.all import *
>>> t = '"%s"'%Integer(10)**Integer(10000)   # ten thousand character string.
>>> a = mathematica(t)         # optional - mathematica
>>> a = mathematica.eval(t)    # optional - mathematica
```

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

```
>>> from sage.all import *
>>> # optional - mathematica
>>> x = mathematica(pi/Integer(2))
>>> print(x)
         Pi
         --
         2
>>> loads(dumps(x)) == x
True
>>> n = x.N(Integer(50))
>>> print(n)
              1.5707963267948966192313216916397514420985846996876
>>> loads(dumps(n)) == n
True
```

26.4 Complicated translations

The `mobj.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)
```

```
>>> from sage.all import *
>>> m = mathematica('NewFn[x]')        # optional - mathematica
>>> m._sage_(locals={('NewFn', Integer(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
```

```
>>> from sage.all import *
>>> def math_bessel_K(nu, x):
...     return mathematica(nu).BesselK(x).N(Integer(20))
>>> math_bessel_K(Integer(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
```

(continues on next page)

(continued from previous page)

```
Mathematica
sage: slist3 = mlist.sage(); slist3          # optional - mathematica
[[1, 2], 3.00000000000000, I + 4]
```

```
>>> from sage.all import *
>>> slist = [[Integer(1), Integer(2)], RealNumber('3.'), Integer(4) + I]
>>> mlist = mathematica(slist); mlist      # optional - mathematica
{{1, 2}, 3., 4 + I}
>>> slist2 = list(mlist); slist2           # optional - mathematica
[{1, 2}, 3., 4 + I]
>>> slist2[Integer(0)]                              # optional - mathematica
{1, 2}
>>> slist2[Integer(0)].parent()                     # optional - mathematica
Mathematica
>>> slist3 = mlist.sage(); slist3          # optional - mathematica
[[1, 2], 3.00000000000000, I + 4]
```

```
sage: mathematica('10.^80')      # optional - mathematica
1.*^80
sage: mathematica('10.^80').sage()   # optional - mathematica
1.00000000000000e80
```

```
>>> from sage.all import *
>>> mathematica('10.^80')      # optional - mathematica
1.*^80
>>> mathematica('10.^80').sage()   # optional - mathematica
1.00000000000000e80
```

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: `ExtraTabCompletion`, *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
"/"
```

```
>>> from sage.all import *
>>> mathematica.chdir('/')            # optional - mathematica
```

(continues on next page)

(continued from previous page)

```
>>> 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: `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
```

```
>>> from sage.all import *
>>> mathematica('Pi').n(Integer(10))     # optional -- mathematica
3.1
>>> mathematica('Pi').n()       # optional -- mathematica
3.14159265358979
>>> mathematica('Pi').n(digits=Integer(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
```

```
>>> from sage.all import *
>>> P = mathematica('Plot[Sin[x],{x,-2Pi,4Pi}]')   # optional - mathematica
>>> filename = tmp_filename()                      # optional - mathematica
>>> P.save_image(filename, ImageSize=Integer(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]
```

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

The following example starts a Mathematica frontend to do the rendering (Issue #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
```

```
>>> from sage.all import *
>>> P = mathematica('Plot[Sin[x],{x,-2Pi,4Pi}]')   # optional - mathematica
>>> show(P)                                        # optional - mathematica␣
↪mathematicafrontend
>>> P.show(ImageSize=Integer(800))                         # optional -␣
↪mathematica mathematicafrontend
```

str()

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

Bases: *ExpectFunction*

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

Bases: *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` – boolean (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]']
```

```
>>> from sage.all import *
>>> from sage.interfaces.mathematica import request_wolfram_alpha
>>> from sage.interfaces.mathematica import parse_moutput_from_json
>>> page_data = request_wolfram_alpha('integrate Sin[x]') # optional internet
>>> 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]}']
```

```
>>> from sage.all import *
>>> page_data = request_wolfram_alpha('Sin[x]')            # optional internet
>>> L = parse_moutput_from_json(page_data)                 # optional internet
>>> 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` – boolean (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']
```

```
>>> from sage.all import *
>>> from sage.interfaces.mathematica import request_wolfram_alpha
>>> page_data = request_wolfram_alpha('integrate Sin[x]')      # optional internet
>>> [str(a) for a in sorted(page_data.keys())]                 # optional internet
['queryresult']
>>> [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',
```

(continues on next page)

(continued from previous page)

```
'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)
```

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

CHAPTER

TWENTYSEVEN

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)
```

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

```
>>> from sage.all import *
>>> # optional - mathics
>>> mobj = mathics(x**Integer(2)-Integer(1))
>>> mobj2 = mobj.Factor(); mobj2
(-1 + x) (1 + x)
>>> mobj2.parent()
Mathics
>>> sobj = mobj2.sage(); sobj
(x + 1)*(x - 1)
>>> sobj.parent()
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}
```

```
>>> from sage.all import *
>>> mathics('Factor[x^2-1]')          # optional - mathics
(-1 + x) (1 + x)
>>> 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]'
```

```
>>> from sage.all import *
>>> mathics.eval('x^2 - 1')             # optional - mathics
'-1 + x ^ 2'
```

(continues on next page)

(continued from previous page)

```
>>> repr(mathics('Range[3]'))          # optional - mathics
'{1, 2, 3}'
>>> 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.

27.1 Tutorial

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

27.1.1 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
```

```
>>> from sage.all import *
>>> m = mathics
>>> a = m(Integer(1)) + m(Integer(1)); a                # optional - mathics
2
>>> a.parent()                        # optional - mathics
Mathics
>>> m('1+1')                          # optional - mathics
2
>>> m(Integer(3))**m(Integer(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
```

```
>>> from sage.all import *
>>> m = mathics
>>> m(Integer(2)).Plus(m(Integer(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
```

```
>>> from sage.all import *
>>> m(Integer(7)).Times(m(Integer(2)).Plus(m(Integer(3))))        # optional - mathics
35
>>> m('7(2+3)')                        # optional - mathics
35
```

27.1.2 Some typical input

We solve an equation and a system of two equations:

```
sage: # optional - mathics
sage: eqn = mathics('3x + 5 == 14')
sage: eqn
5 + 3 x == 14
sage: eqn.Solve('x')
{{x -> 3}}
sage: sys = mathics('{x^2 - 3y == 3, 2x - y == 1}')
sage: print(sys)
{x ^ 2 - 3 y == 3, 2 x - y == 1}
sage: sys.Solve('{x, y}')
{{x -> 0, y -> -1}, {x -> 6, y -> 11}}
```

```
>>> from sage.all import *
>>> # optional - mathics
>>> eqn = mathics('3x + 5 == 14')
>>> eqn
5 + 3 x == 14
>>> eqn.Solve('x')
{{x -> 3}}
>>> sys = mathics('{x^2 - 3y == 3, 2x - y == 1}')
>>> print(sys)
{x ^ 2 - 3 y == 3, 2 x - y == 1}
>>> sys.Solve('{x, y}')
{{x -> 0, y -> -1}, {x -> 6, y -> 11}}
```

27.1.3 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
```

```
>>> from sage.all import *
>>> c = m(Integer(5))                                  # optional - mathics
>>> print(m('b + c x'))                    # optional - mathics
              b + c x
>>> 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}}
```

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

27.1.4 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
```

```
>>> from sage.all import *
>>> m = mathics
>>> _ = mathics('f[p_] = p^2');            # optional - mathics
>>> m('f[9]')                              # optional - mathics
81
```

27.1.5 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}
```

```
>>> from sage.all import *
>>> eqn = mathics('Exp[x] - 3x == 0') # optional - mathics
```

(continues on next page)

(continued from previous page)

```
>>> eqn.FindRoot(['x', Integer(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)')
1.5121345516578424738967396780720387046
```

```
>>> from sage.all import *
>>> gp('solve(x=1,2,exp(x)-3*x)')
1.5121345516578424738967396780720387046
```

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}}
```

```
>>> from sage.all import *
>>> mathics('FindMinimum[x^3 - 6x^2 + 11x - 5, {x,3}]')  # not tested (since not␣
↪supported, so far)
{0.6150998205402516, {x -> 2.5773502699629733}}
>>> f = mathics('x^3 - 6x^2 + 11x - 5')                  # optional - mathics
>>> f.FindMinimum(['x', Integer(3)])                              # not tested (since␣
↪not supported, so far)
{0.6150998205402516, {x -> 2.5773502699629733}}
```

27.1.6 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)
```

```
>>> from sage.all import *
>>> R = PolynomialRing(ZZ, names=('x',)); (x,) = R._first_ngens(1)
>>> f = (x**Integer(100)+Integer(17)*x+Integer(5))*(x**Integer(100)-
↪Integer(5)*x+Integer(20))
```

(continues on next page)

(continued from previous page)

```
>>> f
x^200 + 12*x^101 + 25*x^100 - 85*x^2 + 315*x + 100
>>> g = mathics(str(f))                     # optional - mathics
>>> print(g)                                # optional - mathics
100 + 315 x - 85 x ^ 2 + 25 x ^ 100 + 12 x ^ 101 + x ^ 200
>>> g                                       # optional - mathics
100 + 315 x - 85 x ^ 2 + 25 x ^ 100 + 12 x ^ 101 + x ^ 200
>>> 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)
```

```
>>> from sage.all import *
>>> f = mathics('x^6 + (-y - 2)*x^5 + (y^3 + 2*y)*x^4 - y^4*x^3')  # optional -␣
↪mathics
>>> print(f.Factor())                       # optional - mathics
x ^ 3 (x - y) (-2 x + x ^ 2 + y ^ 3)
```

We factor an integer:

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

```
>>> from sage.all import *
>>> # optional - mathics
>>> n = mathics(Integer(2434500))
>>> n.FactorInteger()
{{2, 2}, {3, 2}, {5, 3}, {541, 1}}
>>> n = mathics(Integer(2434500))
>>> F = n.FactorInteger(); F
{{2, 2}, {3, 2}, {5, 3}, {541, 1}}
>>> F[Integer(1)]
{2, 2}
>>> F[Integer(4)]
{541, 1}
```

27.2 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
```

```
>>> from sage.all import *
>>> t = '"%s"'%Integer(10)**Integer(10000)   # ten thousand character string.
>>> a = mathics(t)        # optional - mathics
>>> a = mathics.eval(t)   # optional - mathics
```

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

```
>>> from sage.all import *
>>> # optional - mathics
>>> x = mathics(pi/Integer(2))
>>> print(x)
Pi / 2
>>> loads(dumps(x)) == x
True
>>> n = x.N(Integer(50))
>>> print(n)
              1.5707963267948966192313216916397514420985846996876
>>> loads(dumps(n)) == n
True
```

27.4 Complicated translations

The `mobj.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)
```

```
>>> from sage.all import *
>>> x = var('x')
>>> m = mathics('NewFn[x]')                       # optional - mathics
>>> 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
```

```
>>> from sage.all import *
>>> def math_bessel_K(nu, x):
...     return mathics(nu).BesselK(x).N(Integer(20))
>>> math_bessel_K(Integer(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], 3.00000000000000, 4.00000000000000 + 1.00000000000000*I]
```

```
>>> from sage.all import *
>>> slist = [[Integer(1), Integer(2)], RealNumber('3.'), Integer(4) + I]
>>> mlist = mathics(slist); mlist          # optional - mathics
```

(continues on next page)

(continued from previous page)

```
{{1, 2}, 3., 4 + I}
>>> slist2 = list(mlist); slist2          # optional - mathics
[{1, 2}, 3., 4 + I]
>>> slist2[Integer(0)]                             # optional - mathics
{1, 2}
>>> slist2[Integer(0)].parent()                    # optional - mathics
Mathics
>>> slist3 = mlist.sage(); slist3          # optional - mathics
[[1, 2], 3.00000000000000, 4.00000000000000 + 1.00000000000000*I]
```

```
sage: mathics('10.^80')          # optional - mathics
1.×10^80
sage: mathics('10.^80').sage()   # optional - mathics
1.00000000000000e80
```

```
>>> from sage.all import *
>>> mathics('10.^80')          # optional - mathics
1.×10^80
>>> mathics('10.^80').sage()   # optional - mathics
1.00000000000000e80
```

AUTHORS:

- Sebastian Oehms (2021): first version from a copy of the Mathematica interface (see Issue #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: *Interface*

Interface to the Mathics interpreter.

Implemented according to the Mathematica interface but avoiding Pexpect functionality.

EXAMPLES:

```
sage: # optional - mathics
sage: t = mathics('Tan[I + 0.5]')
sage: t.parent()
Mathics
sage: ts = t.sage()
sage: ts.parent()
Complex Field with 53 bits of precision
sage: t == mathics(ts)
True
sage: mtan = mathics.Tan
sage: mt = mtan(I+1/2)
sage: mt == t
True
sage: u = mathics(I+1/2)
sage: u.Tan() == mt
True
```

```
>>> from sage.all import *
>>> # optional - mathics
>>> t = mathics('Tan[I + 0.5]')
>>> t.parent()
Mathics
>>> ts = t.sage()
>>> ts.parent()
Complex Field with 53 bits of precision
>>> t == mathics(ts)
True
>>> mtan = mathics.Tan
>>> mt = mtan(I+Integer(1)/Integer(2))
>>> mt == t
True
>>> u = mathics(I+Integer(1)/Integer(2))
>>> u.Tan() == mt
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
/
```

```
>>> from sage.all import *
>>> mathics.chdir('/')              # optional - mathics
>>> 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]
```

(continues on next page)

(continued from previous page)

```
Out[1]= 0.479426

Goodbye!

sage:
```

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

```
>>> from sage.all import *
>>> 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
'2 x + E'
```

```
>>> from sage.all import *
>>> mathics.set('u', '2*x +E')          # optional - mathics
>>> mathics.get('u')                    # optional - mathics
'2 x + E'
```

help(*cmd*, *long=False*)

Return the Mathics documentation of the given command.

EXAMPLES:

```
sage: mathics.help('Sin')                      # optional - mathics
'sine function\n'

sage: print(_)                                 # optional - mathics
sine function


sage: print(mathics.help('Sin', long=True)) # optional - mathics
sine function

Attributes[Sin] = {Listable, NumericFunction, Protected}


sage: print(mathics.Factorial.__doc__)   # optional - mathics
factorial


sage: u = mathics('Pi')                  # optional - mathics
sage: print(u.Cos.__doc__)               # optional - mathics
cosine function
```

```
>>> from sage.all import *
>>> mathics.help('Sin')                        # optional - mathics
'sine function\n'

>>> print(_)                                   # optional - mathics
sine function
<BLANKLINE>

>>> print(mathics.help('Sin', long=True)) # optional - mathics
sine function
<BLANKLINE>
Attributes[Sin] = {Listable, NumericFunction, Protected}
<BLANKLINE>

>>> print(mathics.Factorial.__doc__)   # optional - mathics
factorial
<BLANKLINE>

>>> u = mathics('Pi')                  # optional - mathics
>>> print(u.Cos.__doc__)               # optional - mathics
cosine function
<BLANKLINE>
```

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
```

```
>>> from sage.all import *
>>> mathics.set('u', '2*x +E')          # optional - mathics
>>> bool(mathics('u').sage() == Integer(2)*x+e) # optional - mathics
True
```

class sage.interfaces.mathics.**MathicsElement**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: ExtraTabCompletion, *InterfaceElement*

Element class of the Mathics interface.

Its instances are usually constructed via the instance call of its parent. It wrapes the Mathics library for this object. In a session Mathics methods can be obtained using tab completion.

EXAMPLES:

```
sage: # optional - mathics
sage: me=mathics(e); me
E
sage: type(me)
<class 'sage.interfaces.mathics.MathicsElement'>
sage: P = me.parent(); P
Mathics
sage: type(P)
<class 'sage.interfaces.mathics.Mathics'>
```

```
>>> from sage.all import *
>>> # optional - mathics
>>> me=mathics(e); me
E
>>> type(me)
<class 'sage.interfaces.mathics.MathicsElement'>
>>> P = me.parent(); P
Mathics
>>> type(P)
<class 'sage.interfaces.mathics.Mathics'>
```

Access to the Mathics expression objects:

```
sage: # optional - mathics
sage: res = me._mathics_result
sage: type(res)
<class 'mathics.core.evaluation.Result'>
sage: expr = res.last_eval; expr
<Symbol: System`E>
sage: type(expr)
<class 'mathics.core.symbols.Symbol'>
```

```
>>> from sage.all import *
>>> # optional - mathics
```

(continues on next page)

(continued from previous page)

```
>>> res = me._mathics_result
>>> type(res)
<class 'mathics.core.evaluation.Result'>
>>> expr = res.last_eval; expr
<Symbol: System`E>
>>> type(expr)
<class 'mathics.core.symbols.Symbol'>
```

Applying Mathics methods:

```
sage: # optional - mathics
sage: me.to_sympy()
E
sage: me.get_name()
'System`E'
sage: me.is_inexact()
False
```

```
>>> from sage.all import *
>>> # optional - mathics
>>> me.to_sympy()
E
>>> me.get_name()
'System`E'
>>> me.is_inexact()
False
```

Conversion to Sage:

```
sage: bool(me.sage() == e)               # optional - mathics
True
```

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

```
>>> from sage.all import *
>>> mathics('Pi').n(Integer(10))     # optional -- mathics
```

(continues on next page)

(continued from previous page)

```
3.1
>>> mathics('Pi').n()       # optional -- mathics
3.14159265358979
>>> mathics('Pi').n(digits=Integer(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
```

```
>>> from sage.all import *
>>> P = mathics('Plot[Sin[x],{x,-2Pi,4Pi}]')    # optional - mathics
>>> filename = tmp_filename()                    # optional - mathics
>>> P.save_image(filename, ImageSize=Integer(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
```

```
>>> from sage.all import *
>>> Q = mathics('Sin[x Cos[y]]/Sqrt[1-x^2]')    # optional - mathics
>>> show(Q)                                      # optional - mathics
Sin[x Cos[y]] / Sqrt[1 - x ^ 2]
```

(continues on next page)

(continued from previous page)

```
>>> P = mathics('Plot[Sin[x],{x,-2Pi,4Pi}]')    # optional - mathics
>>> show(P)                                     # optional - mathics
>>> P.show(ImageSize=Integer(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.

In[1]:= Sin[0.5]
Out[1]= 0.479426

Goodbye!
```

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

```
>>> from sage.all import *
>>> sage.interfaces.mathics.reduce_load('Denominator[a / b]')  # optional --␣
↪mathics
b
```

CHAPTER

TWENTYEIGHT

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'
```

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

```
>>> from sage.all import *
>>> a = matlab(Integer(10))                        # optional - matlab
>>> a**Integer(10)                                 # optional - matlab
   1.0000e+10
```

AUTHORS:

- William Stein (2006-10-11)

28.1 Tutorial

EXAMPLES:

```
sage: # optional - matlab
sage: matlab('4+10')
14
sage: matlab('date')
18-Oct-2006
sage: matlab('5*10 + 6')
56
sage: matlab('(6+6)/3')
```

(continues on next page)

(continued from previous page)

```
4
sage: matlab('9')^2
81
sage: a = matlab(10); b = matlab(20); c = matlab(30)
sage: avg = (a+b+c)/3 ; avg
20
sage: parent(avg)
Matlab
```

```
>>> from sage.all import *
>>> # optional - matlab
>>> matlab('4+10')
14
>>> matlab('date')
18-Oct-2006
>>> matlab('5*10 + 6')
56
>>> matlab('(6+6)/3')
4
>>> matlab('9')**Integer(2)
81
>>> a = matlab(Integer(10)); b = matlab(Integer(20)); c = matlab(Integer(30))
>>> avg = (a+b+c)/Integer(3) ; avg
20
>>> parent(avg)
Matlab
```

```
sage: # optional - matlab
sage: my_scalar = matlab('3.1415')
sage: my_scalar
3.1415
sage: my_vector1 = matlab('[1,5,7]')
sage: my_vector1
1     5     7
sage: my_vector2 = matlab('[1;5;7]')
sage: my_vector2
1
5
7
sage: my_vector1 * my_vector2
75
```

```
>>> from sage.all import *
>>> # optional - matlab
>>> my_scalar = matlab('3.1415')
>>> my_scalar
3.1415
>>> my_vector1 = matlab('[1,5,7]')
>>> my_vector1
1     5     7
>>> my_vector2 = matlab('[1;5;7]')
```

(continues on next page)

(continued from previous page)

```
>>> my_vector2
1
5
7
>>> my_vector1 * my_vector2
75
```

```
sage: # optional - matlab
sage: row_vector1 = matlab('[1 2 3]')
sage: row_vector2 = matlab('[3 2 1]')
sage: matrix_from_row_vec = matlab('[%s; %s]'%(row_vector1.name(), row_vector2.
↪name()))
sage: matrix_from_row_vec
1     2     3
3     2     1
```

```
>>> from sage.all import *
>>> # optional - matlab
>>> row_vector1 = matlab('[1 2 3]')
>>> row_vector2 = matlab('[3 2 1]')
>>> matrix_from_row_vec = matlab('[%s; %s]'%(row_vector1.name(), row_vector2.name()))
>>> matrix_from_row_vec
1     2     3
3     2     1
```

```
sage: # optional - matlab
sage: column_vector1 = matlab('[1;3]')
sage: column_vector2 = matlab('[2;8]')
sage: matrix_from_col_vec = matlab('[%s %s]'%(column_vector1.name(), column_vector2.
↪name()))
sage: matrix_from_col_vec
1     2
3     8
```

```
>>> from sage.all import *
>>> # optional - matlab
>>> column_vector1 = matlab('[1;3]')
>>> column_vector2 = matlab('[2;8]')
>>> matrix_from_col_vec = matlab('[%s %s]'%(column_vector1.name(), column_vector2.
↪name()))
>>> matrix_from_col_vec
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
```

```
>>> from sage.all import *
>>> my_matrix = matlab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')     # optional -␣
↪matlab
>>> 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
```

```
>>> from sage.all import *
>>> combined_matrix = matlab('[%s, %s]'%(my_matrix.name(), my_matrix.name()))      ␣
↪                                 # optional - matlab
>>> 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
```

```
>>> from sage.all import *
>>> tm = matlab('0.5:2:10')                                  # optional - matlab
>>> tm                                                       # optional - matlab
0.5000    2.5000    4.5000    6.5000    8.5000
```

```
sage: # optional - matlab
sage: my_vector1 = matlab('[1,5,7]')
sage: my_vector1(1)
1
sage: my_vector1(2)
5
sage: my_vector1(3)
7
```

```
>>> from sage.all import *
>>> # optional - matlab
>>> my_vector1 = matlab('[1,5,7]')
>>> my_vector1(Integer(1))
1
>>> my_vector1(Integer(2))
5
```

(continues on next page)

(continued from previous page)

```
>>> my_vector1(Integer(3))
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
```

```
>>> from sage.all import *
>>> my_matrix = matlab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')      # optional -␣
↪matlab
>>> my_matrix(Integer(3),Integer(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
```

```
>>> from sage.all import *
>>> my_matrix = matlab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')     # optional -␣
↪matlab
>>> my_matrix.set(Integer(2),Integer(3), Integer(1999))                          #␣
↪optional - matlab
>>> 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: *Expect*

Interface to the Matlab interpreter.

EXAMPLES:

```
sage: # optional - matlab
sage: a = matlab('[ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]')
sage: b = matlab('[ 1; 3; 13]')
sage: c = a * b
sage: print(c)
```

(continues on next page)

(continued from previous page)

```
 30
122
505
```

```
>>> from sage.all import *
>>> # optional - matlab
>>> a = matlab('[ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]')
>>> b = matlab('[ 1; 3; 13]')
>>> c = a * b
>>> print(c)
 30
122
505
```

chdir(*directory*)

Change MATLAB's current working directory.

EXAMPLES:

```
sage: matlab.chdir('/')           # optional - matlab
sage: matlab.pwd()                # optional - matlab
/
```

```
>>> from sage.all import *
>>> matlab.chdir('/')           # optional - matlab
>>> 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'
```

```
>>> from sage.all import *
>>> s = matlab.eval('a = 2') # optional - matlab
>>> matlab.get('a')               # optional - matlab
'     2'
```

sage2matlab_matrix_string(*A*)

Return a matlab matrix from a Sage matrix.

INPUT:

- `A` – Sage matrix with entries in the rationals or reals

OUTPUT: string that evaluates to a 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]'
```

```
>>> from sage.all import *
>>> M33 = MatrixSpace(QQ,Integer(3),Integer(3))
>>> A = M33([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6),Integer(7),Integer(8),Integer(0)])
>>> 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*)

Return the string s with Matlab's answer prompt removed.

EXAMPLES:

```
sage: s = '\nans =\n\n     2\n'
sage: matlab.strip_answer(s)
'     2'
```

```
>>> from sage.all import *
>>> s = '\nans =\n\n     2\n'
>>> matlab.strip_answer(s)
'     2'
```

version ()

whos ()

class sage.interfaces.matlab.**MatlabElement** (*parent*, *value*, *is_name=False*, *name=None*)

Bases: *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
```

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

```
>>> from sage.all import *
>>> matlab_version()    # random; optional - matlab
'7.2.0.283 (R2006a)'
```

`sage.interfaces.matlab.`**`reduce_load_Matlab`**`()`

CHAPTER

TWENTYNINE

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'>
```

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

```
>>> from sage.all import *
>>> maxima('3 * 5')
15
```

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

```
sage: x,y = SR.var('x,y')
sage: F = maxima.factor('x^5 - y^5')
sage: F # not tested - depends on maxima version
-((y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4))
sage: actual = F.sage()
sage: expected = -(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4)
sage: bool(actual == expected)
True
sage: type(F)
<class 'sage.interfaces.maxima.MaximaElement'>
```

```
>>> from sage.all import *
>>> x,y = SR.var('x,y')
>>> F = maxima.factor('x^5 - y^5')
>>> F # not tested - depends on maxima version
-((y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4))
>>> actual = F.sage()
>>> expected = -(y-x)*(y**Integer(4)+x*y**Integer(3)+x**Integer(2)*y**Integer(2)+x**In
↪teger(3)*y+x**Integer(4))
>>> bool(actual == expected)
True
>>> 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 )
```

```
>>> from sage.all import *
>>> 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: F = maxima('x * y')
sage: repr(F)
'x*y'
sage: F.str()
'x*y'
```

```
>>> from sage.all import *
>>> F = maxima('x * y')
>>> repr(F)
'x*y'
>>> F.str()
'x*y'
```

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 - 1)'))
(x-1)*(x^4+x^3+x^2+x+1)
```

```
>>> from sage.all import *
>>> print(maxima.eval('factor(x^5 - 1)'))
(x-1)*(x^4+x^3+x^2+x+1)
```

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
(y^5+x^5)^2
sage: f.factor()
(y+x)*(y^4-x*y^3+x^2*y^2-x^3*y+x^4)
```

```
>>> from sage.all import *
>>> f = maxima('x^5 + y^5')
>>> f**Integer(2)
(y^5+x^5)^2
>>> 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
```

```
>>> from sage.all import *
>>> maxima('sum(1/x^2, x, 1, 10)')
1968329/1270080
```

29.1 Tutorial

We follow the tutorial at http://maxima.sourceforge.net/docs/intromax/intromax.html.

```
sage: maxima('1/100 + 1/101')
201/10100
```

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

```
>>> from sage.all import *
>>> a = maxima('(1 + sqrt(2))^5'); a
(sqrt(2)+1)^5
>>> a.expand()
29*sqrt(2)+41
```

```
sage: a = maxima('(1 + sqrt(2))^5')
sage: float(a)
82.0121933088197...
sage: a.numer()
82.0121933088197...
```

```
>>> from sage.all import *
>>> a = maxima('(1 + sqrt(2))^5')
>>> float(a)
82.0121933088197...
>>> a.numer()
82.0121933088197...
```

```
sage: maxima.eval('fpprec : 100')
'100'
sage: a.bfloat()
8.20121933088197564152489730020812442785204843859314941221237124017312418754011041266
↪12384955016056b1
```

```
>>> from sage.all import *
>>> maxima.eval('fpprec : 100')
'100'
>>> a.bfloat()
8.20121933088197564152489730020812442785204843859314941221237124017312418754011041266
↪12384955016056b1
```

```
sage: maxima('100!')
93326215443944152681699238856266700490715968264381621468592963895217599993229915608941
↪4639761565182862536979208272237582511852109168640000000000000000000000000
```

```
>>> from sage.all import *
>>> maxima('100!')
93326215443944152681699238856266700490715968264381621468592963895217599993229915608941
↪4639761565182862536979208272237582511852109168640000000000000000000000000
```

```
sage: f = maxima('(x + 3*y + x^2*y)^3')
sage: f.expand()
x^6*y^3+9*x^4*y^3+27*x^2*y^3+27*y^3+3*x^5*y^2+18*x^3*y^2+27*x*y^2 +3*x^4*y+9*x^2*y+x^3
sage: f.subst('x=5/z')
(5/z+(25*y)/z^2+3*y)^3
sage: g = f.subst('x=5/z')
sage: h = g.ratsimp(); h
(27*y^3*z^6+135*y^2*z^5+(675*y^3+225*y)*z^4+(2250*y^2+125)*z^3 +(5625*y^3+1875*y)*z^
↪2+9375*y^2*z+15625*y^3) /z^6
sage: h.factor()
```

(continues on next page)

(continued from previous page)

```
(3*y*z^2+5*z+25*y)^3/z^6
```

```
>>> from sage.all import *
>>> f = maxima('(x + 3*y + x^2*y)^3')
>>> f.expand()
x^6*y^3+9*x^4*y^3+27*x^2*y^3+27*y^3+3*x^5*y^2+18*x^3*y^2+27*x*y^2 +3*x^4*y+9*x^2*y+x^3
>>> f.subst('x=5/z')
(5/z+(25*y)/z^2+3*y)^3
>>> g = f.subst('x=5/z')
>>> h = g.ratsimp(); h
(27*y^3*z^6+135*y^2*z^5+(675*y^3+225*y)*z^4+(2250*y^2+125)*z^3 +(5625*y^3+1875*y)*z^
↪2+9375*y^2*z+15625*y^3) /z^6
>>> h.factor()
(3*y*z^2+5*z+25*y)^3/z^6
```

```
sage: eqn = maxima(['a+b*c=1', 'b-a*c=0', 'a+b=5'])
sage: s = eqn.solve('[a,b,c]'); s
[[a = -...(sqrt(79)*%i-11)/4...,b = (sqrt(79)*%i+9)/4, c = (sqrt(79)*%i+1)/10], [a =␣
↪(sqrt(79)*%i+11)/4,b = -...(sqrt(79)*%i-9)/4..., c = -...(sqrt(79)*%i-1)/10...]]
```

```
>>> from sage.all import *
>>> eqn = maxima(['a+b*c=1', 'b-a*c=0', 'a+b=5'])
>>> s = eqn.solve('[a,b,c]'); s
[[a = -...(sqrt(79)*%i-11)/4...,b = (sqrt(79)*%i+9)/4, c = (sqrt(79)*%i+1)/10], [a =␣
↪(sqrt(79)*%i+11)/4,b = -...(sqrt(79)*%i-9)/4..., c = -...(sqrt(79)*%i-1)/10...]]
```

Here is an example of solving an algebraic equation:

```
sage: maxima('x^2+y^2=1').solve('y')
[y = -sqrt(1-x^2),y = sqrt(1-x^2)]
sage: maxima('x^2 + y^2 = (x^2 - y^2)/sqrt(x^2 + y^2)').solve('y')
[y = -sqrt((...-y^2...-x^2)*sqrt(y^2+x^2)+x^2), y = sqrt((...-y^2...-x^2)*sqrt(y^2+x^
↪2)+x^2)]
```

```
>>> from sage.all import *
>>> maxima('x^2+y^2=1').solve('y')
[y = -sqrt(1-x^2),y = sqrt(1-x^2)]
>>> maxima('x^2 + y^2 = (x^2 - y^2)/sqrt(x^2 + y^2)').solve('y')
[y = -sqrt((...-y^2...-x^2)*sqrt(y^2+x^2)+x^2), y = sqrt((...-y^2...-x^2)*sqrt(y^2+x^
↪2)+x^2)]
```

You can even nicely typeset the solution in latex:

```
sage: latex(s)
\left[ \left[ a=-...{{\sqrt{79}\,i-11}\over{4}}... , b={{...\sqrt{79}\,i+9...}\over{4}
↪} , c={{\sqrt{79}\,i+1}\over{10}} \right]  , \left[ a={{...\sqrt{79}\,i+11}\over{4}}
↪ , b=-...{{\sqrt{79}\,i-9...}\over{4}}... , c=-...{{...\sqrt{79}\,i-1}\over{10}}...␣
↪\right]  \right]
```

```
>>> from sage.all import *
>>> latex(s)
```

(continues on next page)

(continued from previous page)

```
\left[ \left[ a=-...{{\sqrt{79}\,i-11}\over{4}}... , b={{...\sqrt{79}\,i+9...}\over{4}
↪} , c={{\sqrt{79}\,i+1}\over{10}} \right]  , \left[ a={{...\sqrt{79}\,i+11}\over{4}}
↪ , b=-...{{\sqrt{79}\,i-9...}\over{4}}... , c=-...{{...\sqrt{79}\,i-1}\over{10}}...␣
↪\right]  \right]
```

To have the above appear onscreen via `xdvi`, type `view(s)`. (TODO: For OS X should create pdf output and use preview instead?)

```
sage: e = maxima('sin(u + v) * cos(u)^3'); e
cos(u)^3*sin(v+u)
sage: f = e.trigexpand(); f
cos(u)^3*(cos(u)*sin(v)+sin(u)*cos(v))
sage: f.trigreduce()
(sin(v+4*u)+sin(v-2*u))/8+(3*sin(v+2*u)+3*sin(v))/8
sage: w = maxima('3 + k*%i')
sage: f = w^2 + maxima('%e')^w
sage: f.realpart()
%e^3*cos(k)-k^2+9
```

```
>>> from sage.all import *
>>> e = maxima('sin(u + v) * cos(u)^3'); e
cos(u)^3*sin(v+u)
>>> f = e.trigexpand(); f
cos(u)^3*(cos(u)*sin(v)+sin(u)*cos(v))
>>> f.trigreduce()
(sin(v+4*u)+sin(v-2*u))/8+(3*sin(v+2*u)+3*sin(v))/8
>>> w = maxima('3 + k*%i')
>>> f = w**Integer(2) + maxima('%e')**w
>>> f.realpart()
%e^3*cos(k)-k^2+9
```

```
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)
```

```
>>> from sage.all import *
>>> f = maxima('x^3 * %e^(k*x) * sin(w*x)'); f
x^3*%e^(k*x)*sin(w*x)
>>> 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)
>>> 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-
```

(continues on next page)

(continued from previous page)

```
↪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
```

```
>>> from sage.all import *
>>> f = maxima('1/x^2')
>>> f.integrate('x', Integer(1), 'inf')
1
>>> g = maxima('f/sinh(k*x)^4')
>>> g.taylor('x', Integer(0), Integer(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
```

```
>>> from sage.all import *
>>> maxima.taylor('asin(x)','x',Integer(0), Integer(10))
x+x^3/6+(3*x^5)/40+(5*x^7)/112+(35*x^9)/1152
```

29.2 Examples involving matrices

We illustrate computing with the matrix whose i, j entry is i/j, for $i, j = 1, \ldots, 4$.

```
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]]]]
```

```
>>> from sage.all import *
>>> f = maxima.eval('f[i,j] := i/j')
>>> 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])
>>> A.determinant()
0
>>> A.echelon()
matrix([1,1/2,1/3,1/4],[0,0,0,0],[0,0,0,0],[0,0,0,0])
```

(continues on next page)

(continued from previous page)

```
>>> A.eigenvalues()
[[0,4],[3,1]]
>>> 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
```

```
>>> from sage.all import *
>>> B = matrix(QQ, A)
>>> B.echelon_form()
[  1 1/2 1/3 1/4]
[  0   0   0   0]
[  0   0   0   0]
[  0   0   0   0]
>>> B.charpoly('x').factor()
(x - 4) * x^3
```

29.3 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
```

```
>>> from sage.all import *
>>> _ = maxima.eval("f(t) := t*sin(t)")
>>> 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)
```

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

```
>>> from sage.all import *
>>> _ = maxima.eval("f(t) := exp(t)*sin(t)")
>>> 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)
```

```
>>> from sage.all import *
>>> _ = maxima.eval("f(t) := t^5*exp(t)*sin(t)")
>>> 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
>>> 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)
```

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

```
>>> from sage.all import *
>>> 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)")) # not tested - depends on maxima␣
↪version
                         !
                d        !         2
             (- -- (x(t))!     ) + s  laplace(x(t), t, s) - x(0) s
                dt       !
                         !t = 0
```

```
>>> from sage.all import *
>>> print(maxima("laplace(diff(x(t),t,2),t,s)")) # not tested - depends on maxima␣
↪version
                         !
                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.

29.4 Continued Fractions

A continued fraction $a + 1/(b + 1/(c + \cdots))$ is represented in maxima by the list $[a, b, c, \ldots]$.

```
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]
```

```
>>> from sage.all import *
>>> maxima("cf((1 + sqrt(5))/2)")
[1,1,1,1,2]
>>> 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]
```

29.5 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
```

```
>>> from sage.all import *
>>> f = maxima('2*x^(n+4) - x^(n+2)')
>>> g = maxima('4*x^(n+1) + 3*x^n')
>>> 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]
```

```
>>> from sage.all import *
>>> 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)
                        2/n + 3                   2/n + 1
                      %e                        %e
               ----------------------- - -----------------------
                  1/n         1/n            1/n         1/n
               (%e    - 1) (%e    + 1)   (%e    - 1) (%e    + 1)
```

```
>>> from sage.all import *
>>> S = maxima('nusum(exp(1+2*i/n),i,1,n)')
>>> print(S)
                        2/n + 3                   2/n + 1
                      %e                        %e
               ----------------------- - -----------------------
                  1/n         1/n            1/n         1/n
               (%e    - 1) (%e    + 1)   (%e    - 1) (%e    + 1)
```

We formally compute the limit as $n \to \infty$ of $2S/n$ as follows:

```
sage: T = S*maxima('2/n')
sage: T.tlimit('n','inf')
%e^3-%e
```

```
>>> from sage.all import *
>>> T = S*maxima('2/n')
>>> T.tlimit('n','inf')
%e^3-%e
```

29.6 Miscellaneous

Obtaining digits of π:

```
sage: maxima.eval('fpprec : 100')
'100'
sage: maxima(pi).bfloat()
3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628
↪034825342117068b0
```

```
>>> from sage.all import *
>>> maxima.eval('fpprec : 100')
'100'
>>> maxima(pi).bfloat()
3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628
↪034825342117068b0
```

Defining functions in maxima:

```
sage: maxima.eval('fun[a] := a^2')
'fun[a]:=a^2'
sage: maxima('fun[10]')
100
```

```
>>> from sage.all import *
>>> maxima.eval('fun[a] := a^2')
'fun[a]:=a^2'
>>> maxima('fun[10]')
100
```

29.7 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)')`.

29.8 Latex Output

To TeX a maxima object do this:

```
sage: latex(maxima('sin(u) + sinh(v^2)'))
\sinh v^2+\sin u
```

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

```
>>> from sage.all import *
>>> g = maxima('exp(3*%i*x)/(6*%i) + exp(%i*x)/(2*%i) + c')
>>> latex(g)
-...{{i\,e^{3\,i\,x}}\over{6}}...-{{i\,e^{i\,x}}\over{2}}+c
```

29.9 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)
```

```
>>> from sage.all import *
>>> t = '"%s"'%Integer(10)**Integer(10000)   # ten thousand character string.
>>> a = maxima(t)
```

class sage.interfaces.maxima.**Maxima**(*script_subdirectory=None*, *logfile=None*, *server=None*, *init_code=None*)

Bases: *MaximaAbstract*, *Expect*

Interface to the Maxima interpreter.

EXAMPLES:

```
sage: m = Maxima()
sage: m == maxima
False
```

```
>>> from sage.all import *
>>> m = Maxima()
>>> 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'
```

```
>>> from sage.all import *
>>> maxima.set('xxxxx', '2')
>>> maxima.get('xxxxx')
'2'
>>> maxima.clear('xxxxx')
>>> 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'
```

```
>>> from sage.all import *
>>> maxima.set('xxxxx', '2')
>>> 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
(
```

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

```
>>> from sage.all import *
>>> maxima.set('xxxxx', '2')
>>> 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)
```

(continues on next page)

(continued from previous page)

```
1
sage: [m.random(100) for i in range(5)]
[45, 39, 24, 68, 63]
```

```
>>> from sage.all import *
>>> m = Maxima()
>>> m.set_seed(Integer(1))
1
>>> [m.random(Integer(100)) for i in range(Integer(5))]
[45, 39, 24, 68, 63]
```

class sage.interfaces.maxima.**MaximaElement**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: *MaximaAbstractElement*, *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
```

```
>>> from sage.all import *
>>> maxima(Integer(3))
3
>>> maxima(cos(x)+e**Integer(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 )
```

```
>>> from sage.all import *
>>> F = maxima('x^5 - y^5').factor()
>>> 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: *MaximaElement*, *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
```

```
>>> from sage.all import *
>>> maxima.function('x,y','h(x)*y')
h(x)*y
```

sage.interfaces.maxima.**is_MaximaElement**(*x*)

Return `True` if `x` is of type *MaximaElement*.

EXAMPLES:

```
sage: from sage.interfaces.maxima import is_MaximaElement
sage: is_MaximaElement(1)
doctest:...: DeprecationWarning: the function is_MaximaElement is deprecated; use␣
↪isinstance(x, sage.interfaces.abc.MaximaElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
sage: m = maxima(1)
sage: is_MaximaElement(m)
True
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima import is_MaximaElement
>>> is_MaximaElement(Integer(1))
doctest:...: DeprecationWarning: the function is_MaximaElement is deprecated; use␣
↪isinstance(x, sage.interfaces.abc.MaximaElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
>>> m = maxima(Integer(1))
>>> is_MaximaElement(m)
True
```

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
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima import reduce_load_Maxima
>>> 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
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima import reduce_load_Maxima_function
>>> f = maxima.function('x,y','sin(x+y)')
>>> _,args = f.__reduce__()
>>> g = reduce_load_Maxima_function(*args)
>>> g == f
True
```

CHAPTER THIRTY

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: `ExtraTabCompletion`, *`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
```

```
>>> from sage.all import *
>>> m = Maxima()
>>> from sage.interfaces.maxima_abstract import MaximaAbstract
```

(continues on next page)

(continued from previous page)

```
>>> isinstance(m,MaximaAbstract)
True
```

chdir(*dir*)

Change Maxima's current working directory.

INPUT:

- `dir` – string

OUTPUT: none

EXAMPLES:

```
sage: maxima.chdir('/')
```

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

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

OUTPUT: none

EXAMPLES:

```
sage: maxima.console()             # not tested (since we can't)
Maxima 5.46.0 https://maxima.sourceforge.io
using Lisp ECL 21.2.1
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)
```

```
>>> from sage.all import *
>>> maxima.console()              # not tested (since we can't)
Maxima 5.46.0 https://maxima.sourceforge.io
using Lisp ECL 21.2.1
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()     # not tested
  --> Switching to Maxima <--
maxima: 2+2
4
maxima:
  --> Exiting back to Sage <--
```

```
>>> from sage.all import *
>>> maxima.interact()     # not tested
  --> Switching to Maxima <--
maxima: 2+2
4
maxima:
  --> Exiting back to Sage <--
```

cputime(*t=None*)

Return 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
```

```
>>> from sage.all import *
>>> t = maxima.cputime()
>>> _ = maxima.de_solve('diff(y,x,2) + 3*x = y', ['x','y'], [Integer(1),
↪Integer(1),Integer(1)])
>>> maxima.cputime(t) # output random
0.568913
```

de_solve(*de*, *vars*, *ics=None*)

Solve a 1st or 2nd order ordinary differential equation (ODE) in two variables, possibly with initial conditions.

INPUT:

- `de` – string representing the ODE

- `vars` – 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)...
```

```
>>> from sage.all import *
>>> maxima.de_solve('diff(y,x,2) + 3*x = y', ['x','y'], [Integer(1),
↪Integer(1),Integer(1)])
y = 3*x-2*%e^(x-1)
>>> maxima.de_solve('diff(y,x,2) + 3*x = y', ['x','y'])
y = %k1*%e^x+%k2*%e^-x+3*x
>>> maxima.de_solve('diff(y,x) + 3*x = y', ['x','y'])
y = (%c-3*(...-x...-1)*%e^-x)*%e^x
>>> maxima.de_solve('diff(y,x) + 3*x = y', ['x','y'],[Integer(1),Integer(1)])
y = -...%e^-1*(5*%e^x-3*%e*x-3*%e)...
```

de_solve_laplace(*de*, *vars*, *ics=None*)

Solve an ordinary differential equation (ODE) using Laplace transforms.

INPUT:

- `de` – string representing the ODE (e.g., de = "diff(f(x),x,2)=diff(f(x),x)+sin(x)")
- `vars` – list of strings representing the variables (e.g., `vars = ["x","f"]`)
- `ics` – 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
```

```
>>> from sage.all import *
>>> maxima.clear('x'); maxima.clear('f')
>>> maxima.de_solve_laplace("diff(f(x),x,2) = 2*diff(f(x),x)-f(x)", ["x","f"],
↪ [Integer(0),Integer(1),Integer(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
```

(continues on next page)

(continued from previous page)

```
sage: print(f)
                                                  !
                           x   d                  !                          x           x
               f(x) = x %e   (-- (f(x))!       ) - f(0) x %e  + f(0) %e
                               dx                 !
                                                  !x = 0
```

```
>>> from sage.all import *
>>> maxima.clear('x'); maxima.clear('f')
>>> f = maxima.de_solve_laplace("diff(f(x),x,2) = 2*diff(f(x),x)-f(x)", ["x",
↪"f"])
>>> f
f(x) = x*%e^x*('at('diff(f(x),x,1),x = 0))-f(0)*x*%e^x+f(0)*%e^x
>>> print(f)
                                                  !
                           x   d                  !                          x           x
               f(x) = x %e   (-- (f(x))!       ) - f(0) x %e  + f(0) %e
                               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])
...
```

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

```
>>> from sage.all import *
>>> 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
a[0]:1
a[5]
                                      120
a[n]:=n
a[6]
                                       6
a[4]
                                      24
                                     done
```

```
>>> from sage.all import *
>>> maxima.example('arrays')
a[n]:=n*a[n-1]
                                a  := n a
                                 n       n - 1
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 – string with variable names separated by commas
- defn – 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)
```

```
>>> from sage.all import *
>>> f = maxima.function('x', 'sin(x)')
>>> f(RealNumber('3.2'))  # abs tol 2e-16
-0.058374143427579909
>>> f = maxima.function('x,y', 'sin(x)+cos(y)')
>>> f(Integer(2), RealNumber('3.5'))  # abs tol 2e-16
sin(2)-0.9364566872907963
>>> 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))
```

```
>>> from sage.all import *
>>> g = f.integrate('z')
>>> g
(cos(y)+sin(x))*z
>>> g(Integer(1),Integer(2),Integer(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)
```

(continues on next page)

(continued from previous page)

```
sage: float(t(2))
0.9092974268256817
sage: loads(t.dumps())
gamma(x)*sin(x)
```

```
>>> from sage.all import *
>>> an_expr = maxima('sin(x)*gamma(x)')
>>> t = maxima.function('x', an_expr)
>>> t
gamma(x)*sin(x)
>>> t(Integer(2))
 sin(2)
>>> float(t(Integer(2)))
0.9092974268256817
>>> 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>, ...)
...
```

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

```
>>> from sage.all import *
>>> maxima.plot2d('sin(x)','[x,-5,5]')    # not tested
>>> opts = '[gnuplot_term, ps], [gnuplot_out_file, "sin-plot.eps"]'
>>> 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` – string representing a function (such as r=”[x(t),y(t)]”)
- `var` – string representing the variable (such as var = “t”)
- `trange` – [tmin, tmax] are numbers with tmintmax
- `nticks` – integer (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
```

```
>>> from sage.all import *
>>> maxima.plot2d_parametric(["sin(t)","cos(t)"], "t",[-RealNumber('3.1'),
↪RealNumber('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
```

```
>>> from sage.all import *
>>> opts = '[gnuplot_preamble, "set nokey"], [gnuplot_term, ps], [gnuplot_out_
↪file, "circle-plot.eps"]'
>>> maxima.plot2d_parametric(["sin(t)","cos(t)"], "t", [-RealNumber('3.1'),
↪RealNumber('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
```

```
>>> from sage.all import *
>>> maxima.plot2d_parametric(["sin(5*t)","cos(11*t)"], "t", [Integer(0),
↪Integer(2)*pi()], nticks=Integer(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` – 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
```

```
>>> from sage.all import *
>>> maxima.plot3d('1 + x^3 - y^2', '[x,-2,2]', '[y,-2,2]', '[grid,12,12]')
↪# not tested
>>> maxima.plot3d('sin(x)*cos(y)', '[x,-2,2]', '[y,-2,2]', '[grid,30,30]')
↪# not tested
>>> opts = '[gnuplot_term, ps], [gnuplot_out_file, "sin-plot.eps"]'
>>> 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*, *vrange*, *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 = umin…umax, v = vmin…vmax using gnuplot with options.

INPUT:

- `x`, `y`, `z` – 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]
- `vrange` – [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
```

```
>>> from sage.all import *
>>> maxima.plot3d_parametric(["v*sin(u)","v*cos(u)","v"], ["u","v"],[-
↪RealNumber('3.2'),RealNumber('3.2')],[Integer(0),Integer(3)])      # not␣
↪tested
>>> opts = '[gnuplot_term, ps], [gnuplot_out_file, "sin-cos-plot.eps"]'
>>> maxima.plot3d_parametric(["v*sin(u)","v*cos(u)","v"], ["u","v"],[-
↪RealNumber('3.2'),RealNumber('3.2')],[Integer(0),Integer(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
```

```
>>> from sage.all import *
>>> _ = 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);")
>>> maxima.plot3d_parametric(["expr_1","expr_2","expr_3"], ["x","y"],
↪[Integer(0),Integer(6)],[Integer(0),Integer(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
```

```
>>> from sage.all import *
>>> x = "cos(u)*(3 + v*cos(u/2))"
>>> y = "sin(u)*(3 + v*cos(u/2))"
>>> z = "v*sin(u/2)"
>>> maxima.plot3d_parametric([x,y,z],["u","v"],[-RealNumber('3.1'),RealNumber(
↪'3.2')],[-Integer(1)/Integer(10),Integer(1)/Integer(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` – string representing maxima plot2d options

The points are (x1,y1), (x2,y2), etc.

This function requires maxima 5.9.2 or newer.

Note

More that 150 points can sometimes lead to the program hanging. Why?

EXAMPLES:

```
sage: zeta_ptsx = [(pari(1/2 + i*I/10).zeta().real()).precision(1)          #␣
↪needs sage.libs.pari
....:               for i in range(70,150)]
sage: zeta_ptsy = [(pari(1/2 + i*I/10).zeta().imag()).precision(1)          #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.pari
....:                  for i in range(70,150)]
sage: maxima.plot_list(zeta_ptsx, zeta_ptsy)         # not tested            #␣
↪needs sage.libs.pari
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            #␣
↪needs sage.libs.pari
```

```
>>> from sage.all import *
>>> zeta_ptsx = [(pari(Integer(1)/Integer(2) + i*I/Integer(10)).zeta().
↪real()).precision(Integer(1))          # needs sage.libs.pari
...                 for i in range(Integer(70),Integer(150))]
>>> zeta_ptsy = [(pari(Integer(1)/Integer(2) + i*I/Integer(10)).zeta().
↪imag()).precision(Integer(1))          # needs sage.libs.pari
...                 for i in range(Integer(70),Integer(150))]
>>> maxima.plot_list(zeta_ptsx, zeta_ptsy)         # not tested            #␣
↪needs sage.libs.pari
>>> opts='[gnuplot_preamble, "set nokey"], [gnuplot_term, ps], [gnuplot_out_
↪file, "zeta.eps"]'
>>> maxima.plot_list(zeta_ptsx, zeta_ptsy, opts)  # not tested            #␣
↪needs sage.libs.pari
```

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 that 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)            #␣
↪needs sage.libs.pari
....:               for i in range(10)]
sage: zeta_ptsy1 = [(pari(1/2+i*I/10).zeta().imag()).precision(1)            #␣
↪needs sage.libs.pari
....:               for i in range(10)]
sage: maxima.plot_multilist([[zeta_ptsx1,zeta_ptsy1], [xx,y0], [x0,yy]])     #␣
↪not tested
```

(continues on next page)

(continued from previous page)

```
sage: zeta_ptsx1 = [(pari(1/2+i*I/10).zeta().real()).precision(1)           #␣
↪needs sage.libs.pari
....:                for i in range(10,150)]
sage: zeta_ptsy1 = [(pari(1/2+i*I/10).zeta().imag()).precision(1)           #␣
↪needs sage.libs.pari
....:                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]],    #␣
↪not tested
....:                       opts)
```

```
>>> from sage.all import *
>>> xx = [i/RealNumber('10.0') for i in range(-Integer(10),Integer(10))]
>>> yy = [i/RealNumber('10.0') for i in range(-Integer(10),Integer(10))]
>>> x0 = [Integer(0) for i in range(-Integer(10),Integer(10))]
>>> y0 = [Integer(0) for i in range(-Integer(10),Integer(10))]
>>> zeta_ptsx1 = [(pari(Integer(1)/Integer(2)+i*I/Integer(10)).zeta().real()).
↪precision(Integer(1))           # needs sage.libs.pari
...                for i in range(Integer(10))]
>>> zeta_ptsy1 = [(pari(Integer(1)/Integer(2)+i*I/Integer(10)).zeta().imag()).
↪precision(Integer(1))           # needs sage.libs.pari
...                for i in range(Integer(10))]
>>> maxima.plot_multilist([[zeta_ptsx1,zeta_ptsy1], [xx,y0], [x0,yy]])    #␣
↪not tested
>>> zeta_ptsx1 = [(pari(Integer(1)/Integer(2)+i*I/Integer(10)).zeta().real()).
↪precision(Integer(1))           # needs sage.libs.pari
...                for i in range(Integer(10),Integer(150))]
>>> zeta_ptsy1 = [(pari(Integer(1)/Integer(2)+i*I/Integer(10)).zeta().imag()).
↪precision(Integer(1))           # needs sage.libs.pari
...                for i in range(Integer(10),Integer(150))]
>>> maxima.plot_multilist([[zeta_ptsx1,zeta_ptsy1], [xx,y0], [x0,yy]])    #␣
↪not tested
>>> opts='[gnuplot_preamble, "set nokey"]'
>>> maxima.plot_multilist([[zeta_ptsx1,zeta_ptsy1], [xx,y0], [x0,yy]],    #␣
↪not tested
...                       opts)
```

solve_linear(*eqns*, *vars*)

Wraps maxima's linsolve.

INPUT:

- `eqns` – list of m strings; each representing a linear question in m = n variables
- `vars` – 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]
```

```
>>> from sage.all import *
>>> eqns = ["x + z = y","2*a*x - y = 2*a^2","y - 2*z = 2"]
>>> vars = ["x","y","z"]
>>> 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 – 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?
```

```
>>> from sage.all import *
>>> u = maxima.unit_quadratic_integer(Integer(101)); u
a + 10
>>> u.parent()
Number Field in a with defining polynomial x^2 - 101 with a = 10.
↪04987562112089?
>>> u = maxima.unit_quadratic_integer(Integer(13))
>>> u
5*a + 18
>>> 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.

OUTPUT: none

EXAMPLES:

```
sage: maxima.version()  # random
'5.41.0'
```

```
>>> from sage.all import *
>>> maxima.version()  # random
'5.41.0'
```

class sage.interfaces.maxima_abstract.**MaximaAbstractElement**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: `ExtraTabCompletion`, *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'>
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib
>>> xp = maxima(x)
>>> type(xp)
<class 'sage.interfaces.maxima.MaximaElement'>
>>> xl = maxima_lib(x)
>>> 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)
```

```
>>> from sage.all import *
>>> maxima('sqrt(2) + I').comma('numer')
I+1.41421356237309...
>>> 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
sage: maxima('sin(x^2)').diff('x',4)
16*x^4*sin(x^2)-12*sin(x^2)-48*x^2*cos(x^2)
```

```
>>> from sage.all import *
>>> f = maxima('x^2')
>>> f.diff()
2*x
>>> f.diff('x')
2*x
>>> f.diff('x', Integer(2))
2
>>> maxima('sin(x^2)').diff('x',Integer(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
```

```
>>> from sage.all import *
>>> f = maxima('x^2 + 17*y^2')
>>> f.diff('x')
34*y*'diff(y,x,1)+2*x
>>> 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')
```

(continues on next page)

(continued from previous page)

```
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)
```

```
>>> from sage.all import *
>>> f = maxima('x^2')
>>> f.diff()
2*x
>>> f.diff('x')
2*x
>>> f.diff('x', Integer(2))
2
>>> maxima('sin(x^2)').diff('x',Integer(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
```

```
>>> from sage.all import *
>>> f = maxima('x^2 + 17*y^2')
>>> f.diff('x')
34*y*'diff(y,x,1)+2*x
>>> f.diff('y')
34*y
```

dot(*other*)

Implement 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])
```

```
>>> from sage.all import *
>>> A = maxima('matrix ([a1],[a2])')
>>> B = maxima('matrix ([b1, b2])')
>>> A.dot(B)
matrix([a1*b1,a1*b2],[a2*b1,a2*b2])
```

imag()

Return the imaginary part of this Maxima element.

OUTPUT: Maxima real

EXAMPLES:

```
sage: maxima('2 + (2/3)*%i').imag()
2/3
```

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

```
>>> from sage.all import *
>>> maxima('x^2+1').integral()
x^3/3+x
>>> maxima('x^2+ 1 + y^2').integral('y')
y^3/3+x^2*y+y
>>> maxima('x / (x^2+1)').integral()
log(x^2+1)/2
>>> maxima('1/(x^2+1)').integral()
atan(x)
>>> maxima('1/(x^2+1)').integral('x', Integer(0), infinity)
%pi/2
>>> maxima('x/(x^2+1)').integral('x', -Integer(1), Integer(1))
0
```

```
sage: f = maxima('exp(x^2)').integral('x',0,1)
sage: f.sage()
-1/2*I*sqrt(pi)*erf(I)
sage: f.numer()
1.46265174590718...
```

```
>>> from sage.all import *
>>> f = maxima('exp(x^2)').integral('x',Integer(0),Integer(1))
>>> f.sage()
-1/2*I*sqrt(pi)*erf(I)
>>> 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
```

```
>>> from sage.all import *
>>> maxima('x^2+1').integral()
x^3/3+x
>>> maxima('x^2+ 1 + y^2').integral('y')
y^3/3+x^2*y+y
>>> maxima('x / (x^2+1)').integral()
log(x^2+1)/2
>>> maxima('1/(x^2+1)').integral()
atan(x)
>>> maxima('1/(x^2+1)').integral('x', Integer(0), infinity)
%pi/2
>>> maxima('x/(x^2+1)').integral('x', -Integer(1), Integer(1))
0
```

```
sage: f = maxima('exp(x^2)').integral('x',0,1)
sage: f.sage()
-1/2*I*sqrt(pi)*erf(I)
sage: f.numer()
1.46265174590718...
```

```
>>> from sage.all import *
>>> f = maxima('exp(x^2)').integral('x',Integer(0),Integer(1))
>>> f.sage()
-1/2*I*sqrt(pi)*erf(I)
>>> 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, 4.163...e-11, 231, 0)
```

```
>>> from sage.all import *
>>> maxima('exp(-sqrt(x))').nintegral('x',Integer(0),Integer(1))
(0.5284822353142306, 4.163...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)))')
0.5284822353142307136179049193541565302
sage: _ = gp.set_precision(80)
sage: gp('intnum(x=0,1,exp(-sqrt(x)))')
0.52848223531423071361790491935415653021675547587292866196865279321015401702040079
```

```
>>> from sage.all import *
>>> gp('intnum(x=0,1,exp(-sqrt(x)))')
0.5284822353142307136179049193541565302
>>> _ = gp.set_precision(Integer(80))
>>> gp('intnum(x=0,1,exp(-sqrt(x)))')
0.52848223531423071361790491935415653021675547587292866196865279321015401702040079
```

numer()

Return numerical approximation to `self` as a Maxima object.

OUTPUT: Maxima object

EXAMPLES:

```
sage: a = maxima('sqrt(2)').numer(); a
1.41421356237309...
sage: type(a)
<class 'sage.interfaces.maxima.MaximaElement'>
```

```
>>> from sage.all import *
>>> a = maxima('sqrt(2)').numer(); a
1.41421356237309...
>>> 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)
```

```
>>> from sage.all import *
>>> f = maxima('1/((1+x)*(x-1))')
>>> f.partial_fraction_decomposition('x')
```

(continues on next page)

(continued from previous page)

```
1/(2*(x-1))-1/(2*(x+1))
>>> print(f.partial_fraction_decomposition('x'))
                     1           1
                 --------- - ---------
                 2 (x - 1)   2 (x + 1)
```

real()

Return the real part of this Maxima element.

OUTPUT: Maxima real

EXAMPLES:

```
sage: maxima('2 + (2/3)*%i').real()
2
```

```
>>> from sage.all import *
>>> maxima('2 + (2/3)*%i').real()
2
```

str()

Return string representation of this Maxima object.

OUTPUT: string

EXAMPLES:

```
sage: maxima('sqrt(2) + 1/3').str()
'sqrt(2)+1/3'
```

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

```
>>> from sage.all import *
>>> maxima('a^2 + 3*a + b').subst('b=2')
a^2+3*a+2
```

(continues on next page)

(continued from previous page)

```
>>> maxima('a^2 + 3*a + b').subst('a=17')
b+340
>>> maxima('a^2 + 3*a + b').subst('a=17, b=2')
342
```

class sage.interfaces.maxima_abstract.**MaximaAbstractElementFunction**(*parent*, *name*, *defn*, *args*, *latex*)

Bases: *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)
```

```
>>> from sage.all import *
>>> maxima.function('x,y','e^cos(x)')
e^cos(x)
```

arguments(*split=True*)

Return 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: 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()
[]
```

```
>>> from sage.all import *
>>> f = maxima.function('x,y','sin(x+y)')
>>> f.arguments()
['x', 'y']
>>> f.arguments(split=False)
'x,y'
>>> f = maxima.function('', 'sin(x)')
>>> f.arguments()
[]
```

definition()

Return the definition of this Maxima function as a string.

EXAMPLES:

```
sage: f = maxima.function('x,y','sin(x+y)')
sage: f.definition()
'sin(x+y)'
```

```
>>> from sage.all import *
>>> f = maxima.function('x,y','sin(x+y)')
>>> f.definition()
'sin(x+y)'
```

integral(*var*)

Return 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
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> f = maxima.function('x','sin(x)')
>>> f.integral(x)
-cos(x)
>>> f.integral(y)
sin(x)*y
```

integrate(*var*)

Return 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
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> f = maxima.function('x','sin(x)')
>>> f.integral(x)
-cos(x)
>>> 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 5.46.0 https://maxima.sourceforge.io
...
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_abstract import maxima_console
>>> maxima_console()                          # not tested
Maxima 5.46.0 https://maxima.sourceforge.io
...
```

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'
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_abstract import maxima_version
>>> 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
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_abstract import reduce_load_MaximaAbstract_
↪function
>>> f = maxima.function('x,y','sin(x+y)')
>>> _,args = f.__reduce__()
>>> g = reduce_load_MaximaAbstract_function(*args)
>>> g == f
True
```

CHAPTER

THIRTYONE

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
```

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

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import MaximaLib
>>> 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)
```

```
>>> from sage.all import *
>>> x,n,k = var("x","n","k")
>>> sum((-x)**n/(factorial(n)*factorial(n+Integer(3)/Integer(2))),n,Integer(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
```

```
>>> from sage.all import *
>>> maxima_calculus("besselexpand:false")
false
>>> x,n,k = var("x","n","k")
>>> sum((-x)**n/(factorial(n)*factorial(n+Integer(3)/Integer(2))),n,Integer(0),oo)
-1/2*(2*x*cos(2*sqrt(x)) - sqrt(x)*sin(2*sqrt(x)))/(sqrt(pi)*x^2)
>>> maxima_calculus("besselexpand:true")
true
```

The output is parseable (i. e. Issue #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
```

```
>>> from sage.all import *
>>> foo = maxima_calculus('a and (b or c)') ; foo
a and (b or c)
```

(continues on next page)

(continued from previous page)

```
>>> bar = maxima_calculus(foo) ; bar
a and (b or c)
>>> bar == foo
True
```

class sage.interfaces.maxima_lib.**MaximaLib**

Bases: *MaximaAbstract*

Interface to Maxima as a Library.

OUTPUT: Maxima interface as a Library

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import MaximaLib, maxima_lib
sage: isinstance(maxima_lib,MaximaLib)
True
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import MaximaLib, maxima_lib
>>> 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
```

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

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib
>>> maxima_lib.set('xxxxx', '2')
>>> maxima_lib.get('xxxxx')
'2'
>>> maxima_lib.clear('xxxxx')
>>> 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)'
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib
>>> maxima_lib._eval_line('1+1')
'2'
>>> maxima_lib._eval_line('1+1;')
'2'
>>> maxima_lib._eval_line('1+1$')
''
>>> maxima_lib._eval_line('randvar : cos(x)+sin(y)$')
''
>>> 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'
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib
>>> maxima_lib.set('xxxxx', '2')
>>> 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>
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib
>>> 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')
```

(continues on next page)

(continued from previous page)

```
sage: maxima_lib.get('xxxxx')
'2'
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib
>>> maxima_lib.set('xxxxx', '2')
>>> 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: `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)
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib
>>> maxima_lib(Integer(4))
4
>>> 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 )
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib
>>> F = maxima_lib('x^5 - y^5').factor()
>>> 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.

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|)>
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib
>>> maxima_lib(x+cos(Integer(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...]]
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib
>>> sol = maxima_lib(sin(x) == Integer(0)).to_poly_solve(x)
>>> sol.sage()
[[x == pi*z...]]
```

class sage.interfaces.maxima_lib.**MaximaLibElementFunction**(*parent*, *name*, *defn*, *args*, *latex*)

Bases: *MaximaLibElement*, *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)
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib, dummy_integrate
>>> f = maxima_lib('f(x)').integrate('x')
>>> f.ecl()
<ECL: ((%INTEGRATE SIMP) (($F SIMP) $X) $X)>
>>> 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)
```

```
>>> from sage.all import *
>>> f = maxima_lib('f(x)').integrate('x',Integer(0),Integer(10))
>>> f.ecl()
<ECL: ((%INTEGRATE SIMP) (($F SIMP) $X) $X 0 10)>
>>> dummy_integrate(f.ecl())
integrate(f(x), x, 0, 10)
```

sage.interfaces.maxima_lib.**is_MaximaLibElement**(*x*)

Return `True` if `x` is of type *MaximaLibElement*.

EXAMPLES:

```
sage: from sage.interfaces.maxima_lib import maxima_lib, is_MaximaLibElement
sage: is_MaximaLibElement(1)
doctest:...: DeprecationWarning: the function is_MaximaLibElement is deprecated;␣
↪use isinstance(x, sage.interfaces.abc.MaximaLibElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
sage: m = maxima_lib(1)
```

(continues on next page)

(continued from previous page)

```
sage: is_MaximaLibElement(m)
True
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib, is_MaximaLibElement
>>> is_MaximaLibElement(Integer(1))
doctest:...: DeprecationWarning: the function is_MaximaLibElement is deprecated;␣
↪use isinstance(x, sage.interfaces.abc.MaximaLibElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
>>> m = maxima_lib(Integer(1))
>>> is_MaximaLibElement(m)
True
```

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)
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib, max_at_to_sage
>>> a=maxima_lib("'at(f(x,y,z),[x=1,y=2,z=3])")
>>> a
'at(f(x,y,z),[x = 1,y = 2,z = 3])
>>> max_at_to_sage(a.ecl())
f(1, 2, 3)
>>> a=maxima_lib("'at(f(x,y,z),x=1)")
>>> a
'at(f(x,y,z),x = 1)
>>> 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|)>
sage: max_to_sr(c.ecl())
harmonic_number(x, 2)
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib, max_to_sr
>>> c=maxima_lib(harmonic_number(x,Integer(2)))
>>> c.ecl()
<ECL: (($GEN_HARMONIC_NUMBER SIMP) 2 |$_SAGE_VAR_x|)>
>>> 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)
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib, max_to_sr
>>> c = maxima_lib('pochhammer(x,n)')
>>> c.ecl()
<ECL: (($POCHHAMMER SIMP) $X $N)>
>>> 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)
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib, max_to_sr
>>> f = maxima_lib('f(x)')
```

(continues on next page)

(continued from previous page)

```
>>> f.ecl()
<ECL: (($F SIMP) $X)>
>>> 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)'
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib, max_to_string
>>> ecl = maxima_lib(cos(x)).ecl()
>>> 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)
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib, mdiff_to_sage
>>> f = maxima_lib('f(x)').diff('x',Integer(4))
>>> f.ecl()
<ECL: ((%DERIVATIVE SIMP) (($F SIMP) $X) $X 4)>
>>> 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]
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib, mlist_to_sage
>>> L=maxima_lib("[1,2,3]")
>>> L.ecl()
<ECL: ((MLIST SIMP) 1 2 3)>
>>> 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 li[x](y) and 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)
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib, mqapply_to_sage
>>> c = maxima_lib('li[2](3)')
>>> c.ecl()
<ECL: ((MQAPPLY SIMP) (($LI SIMP ARRAY) 2) 3)>
>>> 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)
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import maxima_lib, mrat_to_sage
>>> var('x y z')
(x, y, z)
>>> c = maxima_lib((x+y**Integer(2)+z**Integer(9))/x**Integer(6)+z**Integer(8)/y).
↪rat()
>>> 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)
>>> c.ecl()
<ECL: ((MRAT SIMP (|$_SAGE_VAR_x| |$_SAGE_VAR_y| |$_SAGE_VAR_z|)
...>
>>> 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)>
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import parse_max_string
>>> 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
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import pyobject_to_max
>>> pyobject_to_max(Integer(4))
<ECL: 4>
>>> pyobject_to_max('z')
<ECL: Z>
>>> var('x')
x
>>> 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
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import reduce_load_MaximaLib
>>> 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
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import sage_rat
>>> sage_rat(Integer(1),Integer(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)>
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import sr_to_max
>>> var('x')
x
>>> sr_to_max(x)
<ECL: $X>
>>> sr_to_max(cos(x))
<ECL: ((%COS) $X)>
>>> f = function('f')(x)
>>> sr_to_max(f.diff())
<ECL: ((%DERIVATIVE) (($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'
```

```
>>> from sage.all import *
>>> from sage.interfaces.maxima_lib import stdout_to_string
>>> stdout_to_string('1+1')
''
>>> stdout_to_string('disp(1+1)')
'2\n\n'
```

CHAPTER

THIRTYTWO

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: ExtraTabCompletion, *Expect*

Interface to the MuPAD interpreter.

completions(*string*, *strip=False*)

EXAMPLES:

```
sage: mupad.completions('linal') # optional - mupad
['linalg']
```

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

```
>>> from sage.all import *
>>> mupad.console() #not tested

   *----*    MuPAD Pro 4.0.2 -- The Open Computer Algebra System
```

(continues on next page)

(continued from previous page)

```
 /|   /|
*----* |    Copyright (c)  1997 - 2007  by SciFace Software
| *--|-*                   All rights reserved.
|/   |/
*----*      Licensed to:   ...
```

cputime (*t=None*)

EXAMPLES:

```
sage: t = mupad.cputime() #random, optional - MuPAD
0.11600000000000001
```

```
>>> from sage.all import *
>>> t = mupad.cputime() #random, optional - MuPAD
0.11600000000000001
```

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

EXAMPLES:

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

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

```
>>> from sage.all import *
>>> a = mupad(Integer(1))    # optional - mupad
>>> 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'
```

```
>>> from sage.all import *
>>> mupad.set('a', Integer(4)) # optional - mupad
>>> 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'
```

```
>>> from sage.all import *
>>> mupad.set('a', Integer(4)) # optional - mupad
>>> mupad.get('a').strip() # optional - mupad
'4'
```

class sage.interfaces.mupad.**MupadElement** (*parent*, *value*, *is_name=False*, *name=None*)

Bases: ExtraTabCompletion, *ExpectElement*

class sage.interfaces.mupad.**MupadFunction** (*parent*, *name*)

Bases: ExtraTabCompletion, *ExpectFunction*

class sage.interfaces.mupad.**MupadFunctionElement** (*obj*, *name*)

Bases: ExtraTabCompletion, *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:   ...
```

```
>>> from sage.all import *
>>> from sage.interfaces.mupad import mupad_console
>>> 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
```

```
>>> from sage.all import *
>>> from sage.interfaces.mupad import reduce_load_mupad
>>> reduce_load_mupad()
Mupad
```

CHAPTER THIRTYTHREE

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: 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...
```

```
>>> from sage.all import *
>>> M = Mwrank('-v 0 -l')
>>> 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: *Expect*

Interface to the Mwrank interpreter.

console()

Start the mwrank console.

EXAMPLES:

```
sage: mwrank.console() # not tested: expects console input
Program mwrank: ...
```

```
>>> from sage.all import *
>>> mwrank.console() # not tested: expects console input
Program mwrank: ...
```

eval(*s*, ***kwds*)

Return mwrank's output for the given input.

INPUT:

- `s` – string; a Sage object which when converted to a string gives valid input to `mwrank`. The conversion is done by *validate_mwrank_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 mwrank interface is restarted and the command is retried once.

EXAMPLES:

```
sage: mwrank.eval('12 3 4 5 6')
'Curve [12,3,4,5,6] :...'
sage: mwrank.eval('[12, 3, 4, 5, 6]')
'Curve [12,3,4,5,6] :...'
sage: mwrank.eval([12, 3, 4, 5, 6])
'Curve [12,3,4,5,6] :...'
sage: mwrank.eval((12, 3, 4, 5, 6))
'Curve [12,3,4,5,6] :...'
```

```
>>> from sage.all import *
>>> mwrank.eval('12 3 4 5 6')
'Curve [12,3,4,5,6] :...'
>>> mwrank.eval('[12, 3, 4, 5, 6]')
'Curve [12,3,4,5,6] :...'
>>> mwrank.eval([Integer(12), Integer(3), Integer(4), Integer(5), Integer(6)])
'Curve [12,3,4,5,6] :...'
```

(continues on next page)

(continued from previous page)

```
>>> mwrank.eval((Integer(12), Integer(3), Integer(4), Integer(5), Integer(6)))
'Curve [12,3,4,5,6] :...'
```

sage.interfaces.mwrank.**mwrank_console**()

Start the mwrank console.

EXAMPLES:

```
sage: mwrank_console() # not tested: expects console input
Program mwrank: ...
```

```
>>> from sage.all import *
>>> mwrank_console() # not tested: expects console input
Program mwrank: ...
```

sage.interfaces.mwrank.**validate_mwrank_input**(*s*)

Return a string suitable for mwrank 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)
```

```
>>> from sage.all import *
>>> from sage.interfaces.mwrank import validate_mwrank_input
>>> validate_mwrank_input([Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)])
'[1, 2, 3, 4, 5]'
>>> validate_mwrank_input((-Integer(1),Integer(2),-Integer(3),Integer(4),-
↪Integer(55)))
```

(continues on next page)

(continued from previous page)

```
'[-1, 2, -3, 4, -55]'
>>> validate_mwrank_input([Integer(1),Integer(2),Integer(3),Integer(4)])
Traceback (most recent call last):
...
ValueError: [1, 2, 3, 4] is not valid input to mwrank (should have 5 entries)
>>> validate_mwrank_input([Integer(1),Integer(2),Integer(3),Integer(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
```

```
>>> from sage.all import *
>>> validate_mwrank_input('0 -1 1 -7 6')
'[0,-1,1,-7,6]'
>>> validate_mwrank_input("[0,-1,1,0,0]\n")
'[0,-1,1,0,0]'
>>> validate_mwrank_input('0\t -1\t 1\t 0\t 0\n')
'[0,-1,1,0,0]'
>>> validate_mwrank_input('0 -1 1 -7 ')
Traceback (most recent call last):
...
ValueError: 0 -1 1 -7  is not valid input to mwrank
```

CHAPTER

THIRTYFOUR

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
```

```
>>> from sage.all import *
>>> octave.eval('2+2')    # optional - octave
'ans = 4'

>>> a = octave(Integer(10))        # optional - octave
>>> a**Integer(10)                 # optional - octave
1e+10
```

LOG: - creation (William Stein) - ? (David Joyner, 2005-12-18) - Examples (David Joyner, 2005-01-03)

34.1 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
```

(continues on next page)

(continued from previous page)

```
    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
    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: # optional - octave
sage: octave("airy(3,2)")
4.10068
sage: octave("beta(2,2)")
0.166667
sage: octave("betainc(0.2,2,2)")
0.104
sage: octave("besselh(0,2)")
(0.223891,0.510376)
sage: octave("besselh(0,1)")
(0.765198,0.088257)
sage: octave("besseli(1,2)")
1.59064
sage: octave("besselj(1,2)")
0.576725
sage: octave("besselk(1,2)")
0.139866
sage: octave("erf(0)")
0
sage: octave("erf(1)")
0.842701
sage: octave("erfinv(0.842)")
0.998315
sage: octave("gamma(1.5)")
0.886227
sage: octave("gammainc(1.5,1)")
0.77687
```

```
>>> from sage.all import *
>>> # optional - octave
>>> octave("airy(3,2)")
4.10068
>>> octave("beta(2,2)")
```

(continues on next page)

(continued from previous page)

```
0.166667
>>> octave("betainc(0.2,2,2)")
0.104
>>> octave("besselh(0,2)")
(0.223891,0.510376)
>>> octave("besselh(0,1)")
(0.765198,0.088257)
>>> octave("besseli(1,2)")
1.59064
>>> octave("besselj(1,2)")
0.576725
>>> octave("besselk(1,2)")
0.139866
>>> octave("erf(0)")
0
>>> octave("erf(1)")
0.842701
>>> octave("erfinv(0.842)")
0.998315
>>> octave("gamma(1.5)")
0.886227
>>> octave("gammainc(1.5,1)")
0.77687
```

34.2 Tutorial

EXAMPLES:

```
sage: # optional - octave
sage: octave('4+10')
14
sage: octave('date')
18-Oct-2007
sage: octave('5*10 + 6')
56
sage: octave('(6+6)/3')
4
sage: octave('9')^2
81
sage: a = octave(10); b = octave(20); c = octave(30)
sage: avg = (a+b+c)/3
sage: avg
20
sage: parent(avg)
Octave
```

```
>>> from sage.all import *
>>> # optional - octave
>>> octave('4+10')
14
>>> octave('date')
```

(continues on next page)

(continued from previous page)

```
18-Oct-2007
>>> octave('5*10 + 6')
56
>>> octave('(6+6)/3')
4
>>> octave('9')**Integer(2)
81
>>> a = octave(Integer(10)); b = octave(Integer(20)); c = octave(Integer(30))
>>> avg = (a+b+c)/Integer(3)
>>> avg
20
>>> parent(avg)
Octave
```

```
sage: # optional - octave
sage: my_scalar = octave('3.1415')
sage: my_scalar
3.1415
sage: my_vector1 = octave('[1,5,7]')
sage: my_vector1
1     5     7
sage: my_vector2 = octave('[1;5;7]')
sage: my_vector2
1
5
7
sage: my_vector1 * my_vector2
75
```

```
>>> from sage.all import *
>>> # optional - octave
>>> my_scalar = octave('3.1415')
>>> my_scalar
3.1415
>>> my_vector1 = octave('[1,5,7]')
>>> my_vector1
1     5     7
>>> my_vector2 = octave('[1;5;7]')
>>> my_vector2
1
5
7
>>> my_vector1 * my_vector2
75
```

class sage.interfaces.octave.**Octave**(*maxread=None*, *script_subdirectory=None*, *logfile=None*, *server=None*, *server_tmpdir=None*, *seed=None*, *command=None*)

Bases: *Expect*

Interface to the Octave interpreter.

EXAMPLES:

```
sage: octave.eval("a = [ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]").strip()    # optional -
↪octave
'a =\n\n 1 1 2\n 3 5 8\n 13 21 33'
sage: octave.eval("b = [ 1; 3; 13]").strip()                         # optional -
↪octave
'b =\n\n 1\n 3\n 13'
```

```
>>> from sage.all import *
>>> octave.eval("a = [ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]").strip()    # optional -
↪octave
'a =\n\n 1 1 2\n 3 5 8\n 13 21 33'
>>> octave.eval("b = [ 1; 3; 13]").strip()                         # optional -
↪octave
'b =\n\n 1\n 3\n 13'
```

The following solves the linear equation: a*c = b:

```
sage: octave.eval(r"c=a \ b").strip()          # optional - octave  # abs tol 0.01
'c =\n\n 1\n -0\n 0'
sage: octave.eval("c").strip()                 # optional - octave  # abs tol 0.01
'c =\n\n 1\n -0\n 0'
```

```
>>> from sage.all import *
>>> octave.eval(r"c=a \ b").strip()          # optional - octave  # abs tol 0.01
'c =\n\n 1\n -0\n 0'
>>> octave.eval("c").strip()                 # optional - octave  # abs tol 0.01
'c =\n\n 1\n -0\n 0'
```

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"
```

```
>>> from sage.all import *
>>> octave.set('x', '2') # optional - octave
>>> octave.clear('x')    # optional - octave
>>> 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]
```

```
>>> from sage.all import *
>>> 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(t0) = x0, y(t0) = y0
- `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)
```

```
>>> from sage.all import *
>>> octave.de_system_plot(['x+y','x-y'], [Integer(1),-Integer(1)],␣
↪[Integer(0),Integer(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

$$x' = x + y, x(0) = 1; \qquad y' = x - y, y(0) = -1, \quad \text{for} \quad 0 < t < 2.$$

get (*var*)

Get the value of the variable `var`.

EXAMPLES:

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

```
>>> from sage.all import *
>>> octave.set('x', '2') # optional - octave
>>> 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.
```

```
>>> from sage.all import *
>>> o = Octave()
>>> o._start()    # optional - octave
>>> 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: 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]'
```

```
>>> from sage.all import *
>>> M33 = MatrixSpace(QQ,Integer(3),Integer(3))
>>> A = M33([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6),Integer(7),Integer(8),Integer(0)])
>>> 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'
```

```
>>> from sage.all import *
>>> octave.set('x', '2') # optional - octave
>>> 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]
```

```
>>> from sage.all import *
>>> o = Octave() # optional - octave
>>> o.set_seed(Integer(1)) # optional - octave
1
>>> [o.rand() for i in range(Integer(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` – mxn matrix A with entries in **Q** or **R**
- `b` – m-vector b entries in **Q** or **R** (resp)

OUTPUT: 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, 0.66666700000000001, -3.5236600000000002e-18]
```

```
>>> from sage.all import *
>>> M33 = MatrixSpace(QQ,Integer(3),Integer(3))
>>> A   = M33([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6),Integer(7),Integer(8),Integer(0)])
>>> V3  = VectorSpace(QQ,Integer(3))
>>> b   = V3([Integer(1),Integer(2),Integer(3)])
>>> octave.solve_linear_system(A,b)    # optional - octave (and output is␣
↪slightly random in low order bits)
[-0.33333299999999999, 0.66666700000000001, -3.5236600000000002e-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
```

```
>>> from sage.all import *
>>> v = octave.version()   # optional - octave
>>> v                      # optional - octave; random
'2.13.7'

>>> import re
>>> 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: 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]
```

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

```
>>> from sage.all import *
>>> from sage.interfaces.octave import reduce_load_Octave
>>> reduce_load_Octave()
Octave
```

sage.interfaces.octave.**to_complex**(*octave_string*, *R*)

Helper function to convert octave complex number.

CHAPTER

THIRTYFIVE

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
[[-1.00000000000000*I, 1.00000000000000*I], [1.00000000000000*I, -1.
↪00000000000000*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
[-1.00000000000000*I, 1.00000000000000*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]]
```

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

(continues on next page)

(continued from previous page)

```
>>> R = PolynomialRing(CDF,Integer(2), names=('x', 'y',)); (x, y,) = R._first_
↪ngens(2)
>>> testsys = [x**Integer(2) + Integer(1), x*y - Integer(1)]
>>> phc.mixed_volume(testsys)        # optional -- phc
2
>>> v = phc.blackbox(testsys, R)     # optional -- phc
>>> sols = v.solutions()             # optional -- phc
>>> sols.sort()                      # optional -- phc
>>> sols                             # optional -- phc
[[-1.00000000000000*I, 1.00000000000000*I], [1.00000000000000*I, -1.
↪00000000000000*I]]
>>> sol_dict = v.solution_dicts()    # optional -- phc
>>> x_sols_from_dict = [d[x] for d in sol_dict]     # optional -- phc
>>> x_sols_from_dict.sort(); x_sols_from_dict       # optional -- phc
[-1.00000000000000*I, 1.00000000000000*I]
>>> residuals = [[test_equation.change_ring(CDF).subs(sol) for test_equation in␣
↪testsys] for sol in v.solution_dicts()]  # optional -- phc
>>> 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` – 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
```

```
>>> from sage.all import *
>>> from sage.interfaces.phc import *
>>> R2 = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R2._
↪first_ngens(2)
>>> start_sys = [x**Integer(6)-y**Integer(2),y**Integer(5)-Integer(1)]
>>> sol = phc.blackbox(start_sys, R2)        # optional -- phc
>>> 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` – 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
```

```
>>> from sage.all import *
>>> from sage.interfaces.phc import *
>>> R2 = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) =␣
↪R2._first_ngens(3)
>>> test_sys = [(x+y+z)**Integer(2)-Integer(1),x**Integer(2)-x,y**Integer(2)-
↪Integer(1)]
>>> 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: 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
```

```
>>> from sage.all import *
>>> from sage.interfaces.phc import *
>>> R2 = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R2._
```

(continues on next page)

(continued from previous page)

```
↪first_ngens(2)
>>> start_sys = [x**Integer(6)-y**Integer(2),y**Integer(5)-Integer(1)]
>>> sol = phc.blackbox(start_sys, R2)          # optional -- phc
>>> start_save = sol.save_as_start()           # optional -- phc
>>> end_sys = [x**Integer(7)-Integer(2),y**Integer(5)-x**Integer(2)]         ␣
↪      # optional -- phc
>>> sol_paths = phc.path_track(start_sys, end_sys, R2, saved_start = start_
↪save)  # optional -- phc
>>> 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'>
```

```
>>> from sage.all import *
>>> from sage.interfaces.phc import *
>>> from sage.structure.sage_object import SageObject
>>> R2 = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R2._
↪first_ngens(2)
>>> start_sys = [x**Integer(5)-y**Integer(2),y**Integer(5)-Integer(1)]
>>> sol = phc.blackbox(start_sys, R2)     # optional -- phc
>>> start_save = sol.save_as_start()      # optional -- phc
>>> end_sys = [x**Integer(5)-Integer(25),y**Integer(5)-x**Integer(2)]         ␣
```

(continues on next page)

(continued from previous page)

```
↪  # optional -- phc
>>> testing = phc.plot_paths_2d(start_sys, end_sys, R2)  # optional -- phc
>>> 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` – 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
```

```
>>> from sage.all import *
>>> from sage.interfaces.phc import *
>>> R2 = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R2._
↪first_ngens(2)
>>> start_sys = [x**Integer(6)-y**Integer(2),y**Integer(5)-Integer(1)]
>>> sol = phc.blackbox(start_sys, R2)        # optional -- phc
>>> start_save = sol.save_as_start()         # optional -- phc
>>> end_sys = [x**Integer(7)-Integer(2),y**Integer(5)-x**Integer(2)]        ␣
↪     # optional -- phc
>>> sol = phc.start_from(start_save, end_sys, R2)  # optional -- phc
>>> 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'
```

```
>>> from sage.all import *
>>> from sage.interfaces.phc import phc
>>> R2 = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R2._first_
↪ngens(2)
>>> start_sys = [(x-Integer(1))**Integer(2)+(y-Integer(1))-Integer(1),␣
↪x**Integer(2)+y**Integer(2)-Integer(1)]
>>> sol = phc.blackbox(start_sys, R2)  # optional -- phc
>>> str(sum([x[Integer(0)] for x in sol.solutions()]).
↪real())[Integer(0):Integer(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
sage: classifieds = sol.classified_solution_dicts()         # optional -- phc
sage: str(sum([q[y] for q in classifieds['real']]))[0:3]      # optional -- phc
'2.0'
```

```
>>> from sage.all import *
>>> from sage.interfaces.phc import phc
>>> R = PolynomialRing(CC,Integer(2), names=('x', 'y',)); (x, y,) = R._first_
↪ngens(2)
>>> p_sys = [x**Integer(10)-y,y**Integer(2)-Integer(1)]
>>> sol = phc.blackbox(p_sys,R)         # optional -- phc
>>> classifieds = sol.classified_solution_dicts()         # optional -- phc
>>> str(sum([q[y] for q in classifieds['real']]))[Integer(0):Integer(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
```

```
>>> from sage.all import *
>>> from sage.interfaces.phc import phc
>>> R2 = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R2._
↪first_ngens(2)
>>> start_sys = [x**Integer(3)-y**Integer(2),y**Integer(5)-Integer(1)]
>>> sol = phc.blackbox(start_sys, R2)  # optional -- phc
>>> start_save = sol.save_as_start()   # optional -- phc
>>> end_sys = [x**Integer(7)-Integer(2),y**Integer(5)-x**Integer(2)]
↪# optional -- phc
>>> sol = phc.start_from(start_save, end_sys, R2)  # optional -- phc
>>> 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` – boolean (default: `False`); the default 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: 1.00000000000000, z: -1.00000000000000, x: 1.00000000000000}
```

```
>>> from sage.all import *
>>> from sage.interfaces.phc import *
>>> R = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = R.
↪_first_ngens(3)
>>> fs = [x**Integer(2)-Integer(1),y**Integer(2)-x,z**Integer(2)-y]
>>> sol = phc.blackbox(fs,R)          # optional -- phc
>>> s_list = sol.solution_dicts()     # optional -- phc
>>> s_list.sort()                     # optional -- phc
>>> s_list[Integer(0)]                         # optional -- phc
{y: 1.00000000000000, z: -1.00000000000000, x: 1.00000000000000}
```

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` – boolean (default: `False`); the default 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
```

```
>>> from sage.all import *
>>> from sage.interfaces.phc import *
>>> R2 = PolynomialRing(QQ,Integer(2), names=('x1', 'x2',)); (x1, x2,) = R2._
↪first_ngens(2)
>>> test_sys = [x1**Integer(5)-x1*x2**Integer(2)-Integer(1), x2**Integer(5)-
↪x1*x2-Integer(1)]
>>> sol = phc.blackbox(test_sys, R2)          # optional -- phc
>>> 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']
```

```
>>> from sage.all import *
>>> from sage.interfaces.phc import *
>>> R2 = PolynomialRing(QQ,Integer(2), names=('x1', 'x2',)); (x1, x2,) = R2._
↪first_ngens(2)
>>> test_sys = [x1**Integer(5)-x1*x2**Integer(2)-Integer(1), x2**Integer(5)-
↪x1*x2-Integer(1)]
>>> sol = phc.blackbox(test_sys, R2)            # optional -- phc
>>> 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
```

```
>>> from sage.all import *
>>> from sage.interfaces.phc import *
>>> R2 = PolynomialRing(QQ,Integer(2), names=('x1', 'x2',)); (x1, x2,) = R2._
↪first_ngens(2)
>>> test_sys = [(x1-Integer(2))**Integer(5)-x2, (x2-Integer(1))**Integer(5)-
↪Integer(1)]
>>> sol = phc.blackbox(test_sys, R2)          # optional -- phc
>>> sol_classes = get_classified_solution_dicts(sol.output_file_contents,R2)  #␣
↪optional -- phc
>>> 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: 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'
```

```
>>> from sage.all import *
>>> from sage.interfaces.phc import *
>>> R2 = PolynomialRing(QQ,Integer(2), names=('x1', 'x2',)); (x1, x2,) = R2._
↪first_ngens(2)
>>> test_sys = [(x1-Integer(1))**Integer(5)-x2, (x2-Integer(1))**Integer(5)-
↪Integer(1)]
>>> sol = phc.blackbox(test_sys, R2)             # optional -- phc
>>> test = get_solution_dicts(sol.output_file_contents,R2)  # optional -- phc
>>> str(sum([q[x1].real() for q in test]))[Integer(0):Integer(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']
```

```
>>> from sage.all import *
>>> from sage.interfaces.phc import *
>>> R2 = PolynomialRing(QQ,Integer(2), names=('x1', 'x2',)); (x1, x2,) = R2._
↪first_ngens(2)
>>> test_sys = [(x1-Integer(2))**Integer(5)-x2, (x2-Integer(1))**Integer(5)-
↪Integer(1)]
>>> sol = phc.blackbox(test_sys, R2)             # optional -- phc
>>> get_variable_list(sol.output_file_contents)  # optional -- phc
['x1', 'x2']
```

sage.interfaces.phc.**random**() → x in the interval [0, 1).

CHAPTER

THIRTYSIX

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 Ewgenij 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 *PolymakeJuPyMake*

class sage.interfaces.polymake.**PolymakeAbstract**(*seed=None*)

Bases: ExtraTabCompletion, *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
```

```
>>> from sage.all import *
>>> 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_verbose(3)
sage: p.H_VECTOR                                         # optional - jupymake
polymake: used package ppl
  The Parma Polyhedra Library ...
1 16 40 16 1
sage: set_verbose(0)
sage: p.F_VECTOR                                         # optional - jupymake
20 94 148 74
```

```
>>> from sage.all import *
>>> isinstance(polymake_jupymake, PolymakeAbstract)
```

(continues on next page)

(continued from previous page)

```
True
>>> p = polymake_jupymake.rand_sphere(Integer(4), Integer(20), seed=Integer(5))  ␣
↪    # optional - jupymake
>>> p                                            # optional - jupymake
Random spherical polytope of dimension 4; seed=5...
>>> set_verbose(Integer(3))
>>> p.H_VECTOR                                   # optional - jupymake
polymake: used package ppl
  The Parma Polyhedra Library ...
1 16 40 16 1
>>> set_verbose(Integer(0))
>>> p.F_VECTOR                                   # optional - jupymake
20 94 148 74
```

application(*app*)

Change to a given polymake application.

INPUT:

- `app` – 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: # optional - jupymake
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]])
sage: q.H_VECTOR
1 5 5 1
sage: q.F_VECTOR
8 14 8
sage: q.VERY_AMPLE
true
```

```
>>> from sage.all import *
>>> # optional - jupymake
>>> q = polymake.new_object("Polytope", INEQUALITIES=[[Integer(5),-Integer(4),
↪Integer(0),Integer(1)],[-Integer(3),Integer(0),-Integer(4),Integer(1)],[-
↪Integer(2),Integer(1),Integer(0),Integer(0)],[-Integer(4),Integer(4),
↪Integer(4),-Integer(1)],[Integer(0),Integer(0),Integer(1),Integer(0)],
↪[Integer(8),Integer(0),Integer(0),-Integer(1)],[Integer(1),Integer(0),-
↪Integer(1),Integer(0)],[Integer(3),-Integer(1),Integer(0),Integer(0)]])
>>> q.H_VECTOR
1 5 5 1
>>> q.F_VECTOR
8 14 8
>>> q.VERY_AMPLE
true
```

In the application 'fan', polymake can now compute the normal fan of q and its (primitive) rays:

```
sage: # optional - jupymake
sage: polymake.application('fan')
sage: g = q.normal_fan()
sage: g.RAYS
-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()
-4 0 1
0 -4 1
1 0 0
4 4 -1
0 1 0
0 0 -1
0 -1 0
-1 0 0
```

```
>>> from sage.all import *
>>> # optional - jupymake
>>> polymake.application('fan')
>>> g = q.normal_fan()
>>> g.RAYS
-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
>>> g.RAYS.primitive()
-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):
...
NotImplementedError: Please use polymake_console() function or the .
↪interact() method
```

```
>>> from sage.all import *
>>> polymake.console()
Traceback (most recent call last):
...
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...
```

```
>>> from sage.all import *
>>> polymake.rand_sphere(Integer(4), Integer(30), seed=Integer(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(...)'
```

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

```
>>> from sage.all import *
>>> 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` – string
- `pager` – boolean (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.
```

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

```
>>> from sage.all import *
>>> polymake.help('TRIANGULATION')                          # optional -␣
```

(continues on next page)

(continued from previous page)

```
↪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/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'
```

```
>>> from sage.all import *
>>> polymake.help('Triangulation')      # optional - jupymake
Traceback (most recent call last):
...
PolymakeError: unknown help topic 'Triangulation'
```

new_object(*name*, **args*, ***kwds*)

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: # optional - jupymake
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]])
sage: q.N_VERTICES
4
sage: q.BOUNDED
true
sage: q.VERTICES
1 2 0 4
1 3 0 8
1 2 1 8
1 3 1 8
sage: q.full_typename()
'Polytope<Rational>'
```

```
>>> from sage.all import *
>>> # optional - jupymake
```

(continues on next page)

(continued from previous page)

```
>>> q = polymake.new_object("Polytope<Rational>", INEQUALITIES=[[Integer(4),-
↪Integer(4),Integer(0),Integer(1)],[-Integer(4),Integer(0),-Integer(4),
↪Integer(1)],[-Integer(2),Integer(1),Integer(0),Integer(0)],[-Integer(4),
↪Integer(4),Integer(4),-Integer(1)],[Integer(0),Integer(0),Integer(1),
↪Integer(0)],[Integer(8),Integer(0),Integer(0),-Integer(1)]])
>>> q.N_VERTICES
4
>>> q.BOUNDED
true
>>> q.VERTICES
1 2 0 4
1 3 0 8
1 2 1 8
1 3 1 8
>>> q.full_typename()
'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
```

```
>>> from sage.all import *
>>> c = polymake('cube(3)')                          # optional - jupymake #␣
↪indirect doctest
>>> d = polymake.cube(Integer(3))                             # optional -␣
↪jupymake
```

Equality is, for "big" objects such as polytopes, comparison by identity:

```
sage: c == d                                         # optional - jupymake
False
```

```
>>> from sage.all import *
>>> c == d                                         # optional - jupymake
False
```

However, the list of vertices is equal:

```
sage: c.VERTICES == d.VERTICES                      # optional - jupymake
True
```

```
>>> from sage.all import *
>>> c.VERTICES == d.VERTICES                      # optional - jupymake
True
```

version()

Version of the polymake installation.

EXAMPLES:

```
sage: polymake.version()              # optional - jupymake # random
'4...'
```

```
>>> from sage.all import *
>>> polymake.version()              # optional - jupymake # random
'4...'
```

class sage.interfaces.polymake.**PolymakeElement**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: `ExtraTabCompletion`, *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...
```

```
>>> from sage.all import *
>>> p = polymake.rand_sphere(Integer(4), Integer(20), seed=Integer(5))
↪# optional - jupymake
>>> p.typename()                                       # optional - jupymake
'Polytope'
>>> 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.VERTICES[2][2]                                      # optional - jupymake
1450479926727001/2251799813685248
```

```
>>> from sage.all import *
>>> p.VERTICES[Integer(2)][Integer(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>'
sage: c.VERTICES.full_typename()     # optional - jupymake
'Matrix<Rational, NonSymmetric>'
```

```
>>> from sage.all import *
>>> c = polymake.cube(Integer(4))          # optional - jupymake
>>> c.full_typename()             # optional - jupymake
'Polytope<Rational>'
>>> c.VERTICES.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
```

```
>>> from sage.all import *
>>> p = polymake.rand_sphere(Integer(4), Integer(20), seed=Integer(5))     #␣
↪optional - jupymake
```

Normally, a property would be accessed as follows:

```
sage: p.F_VECTOR                                  # optional - jupymake
20 94 148 74
```

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

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

```
>>> from sage.all import *
>>> p.type                                        # optional - jupymake
Member function 'type' of Polymake::polytope::Polytope__Rational object
>>> p.get_member('type')                          # optional - jupymake
Polytope<Rational>[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...]
```

```
>>> from sage.all import *
>>> 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: # optional - jupymake
sage: c = polymake.cube(2)
sage: c.contains
Member function 'contains' of Polymake::polytope::Polytope__Rational object
sage: V = polymake.new_object('Vector', [1,0,0])
sage: V
1 0 0
sage: c.contains(V)
true
```

```
>>> from sage.all import *
>>> # optional - jupymake
>>> c = polymake.cube(Integer(2))
>>> c.contains
Member function 'contains' of Polymake::polytope::Polytope__Rational object
>>> V = polymake.new_object('Vector', [Integer(1),Integer(0),Integer(0)])
>>> V
1 0 0
>>> c.contains(V)
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 -␣
↪jupymake
Member function 'foo' of Polymake::polytope::Polytope__Rational object
sage: c.get_member_function("foo")()                            # optional -␣
↪jupymake
Traceback (most recent call last):
...
TypeError: Can't locate object method "foo" via package
↪"Polymake::polytope::Polytope__Rational"
```

```
>>> from sage.all import *
>>> c.get_member_function("foo")                              # optional -␣
↪jupymake
Member function 'foo' of Polymake::polytope::Polytope__Rational object
>>> c.get_member_function("foo")()                            # optional -␣
↪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',
```

(continues on next page)

(continued from previous page)

```
...
 'VERTICES_IN_FACETS']
sage: c.list_properties()                        # optional - jupymake
CONE_AMBIENT_DIM, CONE_DIM, FACETS, AFFINE_HULL, VERTICES_IN_FACETS,
BOUNDED...
```

```
>>> from sage.all import *
>>> c = polymake.cube(Integer(4))                          # optional - jupymake
>>> c.known_properties()                         # optional - jupymake
['AFFINE_HULL',
 'BOUNDED',
 'CONE_AMBIENT_DIM',
 'CONE_DIM',
...
 'VERTICES_IN_FACETS']
>>> 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',
...
 'VERTICES_IN_FACETS']
```

```
>>> from sage.all import *
>>> c.F_VECTOR                                   # optional - jupymake
16 32 24 8
>>> c.known_properties()                         # optional - jupymake
['AFFINE_HULL',
 'BOUNDED',
 'COMBINATORIAL_DIM',
 'CONE_AMBIENT_DIM',
 'CONE_DIM',
...
 '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>'
```

(continues on next page)

(continued from previous page)

```
sage: c.VERTICES.qualified_typename()    # optional - jupymake
'common::Matrix<Rational, NonSymmetric>'
```

```
>>> from sage.all import *
>>> c = polymake.cube(Integer(4))                # optional - jupymake
>>> c.qualified_typename()             # optional - jupymake
'polytope::Polytope<Rational>'
>>> c.VERTICES.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.VERTICES.typename()           # optional - jupymake
'Matrix'
```

```
>>> from sage.all import *
>>> c = polymake.cube(Integer(4))            # optional - jupymake
>>> c.typename()                    # optional - jupymake
'Polytope'
>>> c.VERTICES.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: # optional - jupymake
sage: p = polymake.rand_sphere(3, 13, seed=12)
sage: p.typeof()
('Polymake::polytope::Polytope__Rational', 'ARRAY')
sage: p.VERTICES.typeof()
('Polymake::common::Matrix_A_Rational_I_NonSymmetric_Z', 'ARRAY')
sage: p.get_schedule('"F_VECTOR"').typeof()
('Polymake::Core::Scheduler::RuleChain', 'ARRAY')
```

```
>>> from sage.all import *
>>> # optional - jupymake
>>> p = polymake.rand_sphere(Integer(3), Integer(13), seed=Integer(12))
>>> p.typeof()
('Polymake::polytope::Polytope__Rational', 'ARRAY')
```

(continues on next page)

(continued from previous page)

```
>>> p.VERTICES.typeof()
('Polymake::common::Matrix_A_Rational_I_NonSymmetric_Z', 'ARRAY')
>>> p.get_schedule('"F_VECTOR"').typeof()
('Polymake::Core::Scheduler::RuleChain', 'ARRAY')
```

On “small” objects, it just returns empty strings:

```
sage: p.N_VERTICES.typeof()                                          # optional -␣
↪jupymake
('', '')
sage: p.list_properties().typeof()                                   # optional -␣
↪jupymake
('', '')
```

```
>>> from sage.all import *
>>> p.N_VERTICES.typeof()                                          # optional -␣
↪jupymake
('', '')
>>> 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: *InterfaceFunctionElement*

A callable (function or member function) bound to a polymake element.

EXAMPLES:

```
sage: # optional - jupymake
sage: c = polymake.cube(2)
sage: V = polymake.new_object('Vector', [1,0,0])
sage: V
1 0 0
sage: c.contains
Member function 'contains' of Polymake::polytope::Polytope__Rational object
sage: c.contains(V)
true
```

```
>>> from sage.all import *
>>> # optional - jupymake
>>> c = polymake.cube(Integer(2))
>>> V = polymake.new_object('Vector', [Integer(1),Integer(0),Integer(0)])
>>> V
1 0 0
>>> c.contains
Member function 'contains' of Polymake::polytope::Polytope__Rational object
>>> c.contains(V)
true
```

class sage.interfaces.polymake.**PolymakeJuPyMake**(*seed=None*, *verbose=False*)

Bases: 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.:
    $v = new Vector<Int>(1,2,3);
 or
    $v = new Vector<Int>([1,2,3]);
```

```
>>> from sage.all import *
>>> from sage.interfaces.polymake import polymake_jupymake as polymake
>>> type(polymake)
<...sage.interfaces.polymake.PolymakeJuPyMake...
>>> p = polymake.rand_sphere(Integer(4), Integer(20), seed=Integer(5))       #␣
↪optional - jupymake
>>> p                                             # optional - jupymake
Random spherical polytope of dimension 4; seed=5...
>>> set_verbose(Integer(3))
>>> p.H_VECTOR;                                   # optional - jupymake # random
used package ppl
  The Parma Polyhedra Library ...
>>> p.H_VECTOR                                    # optional - jupymake
1 16 40 16 1
```

(continues on next page)

(continued from previous page)

```
>>> set_verbose(Integer(0))
>>> p.F_VECTOR                                    # optional - jupymake
20 94 148 74
>>> 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.:
    $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: # long time, optional - internet jupymake perl_mongodb
sage: L = polymake.db_query({'"_id"': '"F.4D.0047"'},
....:                       db='"LatticePolytopes"',
....:                       collection='"SmoothReflexive"'); L
BigObjectArray
sage: len(L)
1
sage: P = L[0]
sage: sorted(P.list_properties(), key=str)
[..., LATTICE_POINTS_GENERATORS, ..., POINTED, ...]
sage: P.F_VECTOR
20 40 29 9
```

```
>>> from sage.all import *
>>> # long time, optional - internet jupymake perl_mongodb
>>> L = polymake.db_query({'"_id"': '"F.4D.0047"'},
...                       db='"LatticePolytopes"',
...                       collection='"SmoothReflexive"'); L
BigObjectArray
>>> len(L)
1
>>> P = L[Integer(0)]
>>> sorted(P.list_properties(), key=str)
[..., LATTICE_POINTS_GENERATORS, ..., POINTED, ...]
>>> P.F_VECTOR
20 40 29 9
```

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

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
```

```
>>> from sage.all import *
>>> from sage.interfaces.polymake import polymake_jupymake as polymake
>>> p = polymake.cube(Integer(3))              # optional - jupymake  #␣
↪indirect doctest
```

Here we see that remarks printed by polymake are displayed if the verbosity is positive:

```
sage: set_verbose(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 Koeppe, Jesus A. De Loera and others.
  http://www.math.ucdavis.edu/~latte/
27
sage: set_verbose(0)
```

```
>>> from sage.all import *
>>> set_verbose(Integer(1))
>>> 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 Koeppe, Jesus A. De Loera and others.
  http://www.math.ucdavis.edu/~latte/
27
>>> set_verbose(Integer(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...
```

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

```
>>> from sage.all import *
>>> polymake.eval('print 3')          # optional - jupymake
Traceback (most recent call last):
...
SyntaxError: Incomplete polymake command 'print 3'
>>> 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
```

```
>>> from sage.all import *
>>> print(polymake.eval('$tmp="abc";\nprint $tmp;'))  # optional - jupymake
abc
```

When requesting help, polymake sometimes expects 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
```

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

```
>>> from sage.all import *
>>> c = polymake.cube(Integer(15))               # optional - jupymake
>>> polymake.eval('print {}->F_VECTOR;'.format(c.name()), timeout=Integer(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
```

```
>>> from sage.all import *
>>> c                                  # optional - jupymake
cube of dimension 15
>>> 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 ...
...
Ewgenij 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:
```

(continues on next page)

(continued from previous page)

```
4ti2, bliss, cdd, latte, libnormaliz, lrs, permlib, ppl, sketch, sympol, threejs,
↪tikz, topcom, tosimplex
For more details:  show_credits;
polytope >
```

```
>>> from sage.all import *
>>> from sage.interfaces.polymake import polymake_console
>>> polymake_console()        # not tested
Welcome to polymake version ...
...
Ewgenij 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
```

```
>>> from sage.all import *
>>> from sage.interfaces.polymake import reduce_load_Polymake
>>> reduce_load_Polymake()
Polymake
```

CHAPTER

THIRTYSEVEN

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 POVRay

OUTPUT: image is written to the file you specified in outfile

EXAMPLES:

AUTHOR:

Sage interface written by Yi Qiang (yqiang _atNOSPAM_ gmail.com)

POVRay: http://www.povray.org

usage()

CHAPTER

THIRTYEIGHT

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)]
```

```
>>> from sage.all import *
>>> v = [PSage() for _ in range(Integer(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)
```

```
>>> from sage.all import *
>>> w = [x('factor(2^%s-1)'% randint(Integer(250),Integer(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]
```

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

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

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

INPUT:

- 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: *SageElement*

is_locked()

CHAPTER

THIRTYNINE

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
```

```
>>> from sage.all import *
>>> var('a,b,c,d,x,y,z')
(a, b, c, d, x, y, z)
>>> 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
```

```
>>> from sage.all import *
>>> ellipse = Integer(3)*x**Integer(2) + Integer(2)*x*y + y**Integer(2) - x + y -
↪Integer(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]
```

```
>>> from sage.all import *
>>> F = qf.exists(y, ellipse == Integer(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
```

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

```
>>> from sage.all import *
>>> qepcad(qf.exists(x, ellipse == Integer(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
```

```
>>> from sage.all import *
>>> circle = x**Integer(2) + y**Integer(2) - Integer(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 ]
```

```
>>> from sage.all import *
>>> F = qf.exactly_k(Integer(3), y, circle * ellipse == Integer(0)); F
(X3 y)[(3 x^2 + 2 x y + y^2 - x + y - 7) (x^2 + y^2 - 3) = 0]
>>> 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
```

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

```
>>> from sage.all import *
>>> F = qf.and_(ellipse < Integer(0), circle < Integer(0)); F
[3 x^2 + 2 x y + y^2 - x + y - 7 < 0 /\ x^2 + y^2 - 3 < 0]
>>> 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
]
```

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

(continues on next page)

(continued from previous page)

```
sage: qepcad(F)                                           # not tested (random␣
↪order)
x^2 - 3 <= 0 /\ [ x > 0 \/ 2 x^2 - 3 < 0 ]
```

```
>>> from sage.all import *
>>> F = qf.exists(y, qf.and_(circle == Integer(0), x + y > Integer(0))); F
(E y)[x^2 + y^2 - 3 = 0 /\ x + y > 0]
>>> 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
```

```
>>> from sage.all import *
>>> 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. With all-points, `qepcad()` either produces a list of dictionaries for all points where the formula is true, or raises an exception if the formula is true on infinitely many points. With cell-points, `qepcad()` produces a list of dictionaries with one point for each cell where the formula is true. (This means you will have at least one point in each connected component of the solution, although you will often have many more points than that.)

Let us revisit some of the above examples and get some points to play with. We will start by finding a point on our ellipse.

```
sage: p = qepcad(ellipse == 0, solution='any-point'); p   # optional - qepcad
{'x': -1.468501968502953?, 'y': 0.9685019685029527?}
```

```
>>> from sage.all import *
>>> p = qepcad(ellipse == Integer(0), solution='any-point'); p   # optional - qepcad
{'x': -1.468501968502953?, 'y': 0.9685019685029527?}
```

(Note that despite the decimal printing and the question marks, these are really exact numbers.)

We can verify that this point is a solution. To do so, we create a copy of ellipse as a polynomial over **Q** (instead of a symbolic expression).

```
sage: pellipse = QQ['x,y'](ellipse)
sage: pellipse(**p) == 0                             # optional - qepcad
True
```

```
>>> from sage.all import *
>>> pellipse = QQ['x,y'](ellipse)
>>> pellipse(**p) == Integer(0)                              # optional - qepcad
True
```

For cell-points, let us look at points *not* on the ellipse.

```
sage: pts = qepcad(ellipse != 0, solution='cell-points'); pts    # optional - qepcad
[{'x': 4, 'y': 0},
 {'x': 2.468501968502953?, 'y': 1},
 {'x': 2.468501968502953?, 'y': -9},
 {'x': 1/2, 'y': 9},
 {'x': 1/2, 'y': -1},
 {'x': 1/2, 'y': -5},
 {'x': -1.468501968502953?, 'y': 3},
 {'x': -1.468501968502953?, 'y': -1},
 {'x': -3, 'y': 0}]
```

```
>>> from sage.all import *
>>> pts = qepcad(ellipse != Integer(0), solution='cell-points'); pts    # optional -␣
↪qepcad
[{'x': 4, 'y': 0},
 {'x': 2.468501968502953?, 'y': 1},
 {'x': 2.468501968502953?, 'y': -9},
 {'x': 1/2, 'y': 9},
 {'x': 1/2, 'y': -1},
 {'x': 1/2, 'y': -5},
 {'x': -1.468501968502953?, 'y': 3},
 {'x': -1.468501968502953?, 'y': -1},
 {'x': -3, 'y': 0}]
```

For the points here which are in full-dimensional cells, QEPCAD has the freedom to choose rational sample points, and it does so.

And, of course, all these points really are not on the ellipse.

```
sage: [pellipse(**p) != 0 for p in pts]                  # optional - qepcad
[True, True, True, True, True, True, True, True, True]
```

```
>>> from sage.all import *
>>> [pellipse(**p) != Integer(0) for p in pts]                # optional - qepcad
[True, True, True, True, True, True, True, True, True]
```

Finally, for all-points, let us look again at finding vertical lines that intersect the union of the circle and the 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: pts = qepcad(F, solution='all-points'); pts        # optional - qepcad
[{'x': 1.732050807568878?}, {'x': 1.731054913462534?}, {'x': 0.678911384208004?}, {'x
↪': -0.9417727377417167?}, {'x': -1.468193559928821?}, {'x': -1.468501968502953?}]
```

```
>>> from sage.all import *
>>> F = qf.exactly_k(Integer(3), y, circle * ellipse == Integer(0)); F
(X3 y)[(3 x^2 + 2 x y + y^2 - x + y - 7) (x^2 + y^2 - 3) = 0]
>>> pts = qepcad(F, solution='all-points'); pts       # optional - qepcad
[{'x': 1.732050807568878?}, {'x': 1.731054913462534?}, {'x': 0.678911384208004?}, {'x
↪': -0.9417727377417167?}, {'x': -1.468193559928821?}, {'x': -1.468501968502953?}]
```

Since y is bound by the quantifier, the solutions only refer to x.

We can substitute one of these solutions into the original equation:

```
sage: pt = pts[0]                              # optional - qepcad
sage: pcombo = QQ['x,y'](circle * ellipse)
sage: intersections = pcombo(y=polygen(AA, 'y'), **pt); intersections       #␣
↪optional - qepcad
y^4 + 4.464101615137755?*y^3 + 0.2679491924311227?*y^2
```

```
>>> from sage.all import *
>>> pt = pts[Integer(0)]                       # optional - qepcad
>>> pcombo = QQ['x,y'](circle * ellipse)
>>> 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)]
```

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

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

```
>>> from sage.all import *
>>> qe = qepcad(qf.forall(x, x**Integer(2) + b*x + c > Integer(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.
```

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

```
>>> from sage.all import *
>>> qe.go()                                          # optional - qepcad
QEPCAD object has moved to phase 'Before Projection (x)'
>>> qe.go()                                          # optional - qepcad
QEPCAD object has moved to phase 'Before Choice'
>>> 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'
```

```
>>> from sage.all import *
>>> qe.go()                                          # optional - qepcad
```

(continues on next page)

(continued from previous page)

```
4 c - b^2 > 0
>>> 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'
```

```
>>> from sage.all import *
>>> qe = qepcad(circle == Integer(0), interact=True); qe       # optional - qepcad
QEPCAD object in phase 'Before Normalization'
>>> 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
```

(continues on next page)

(continued from previous page)

```
      = 1.7320508076-

Coordinate 1 = 0
             = 0.0000000000
Coordinate 2 = alpha
             = 1.7320508076-
----------------------------------------------------
```

```
>>> from sage.all import *
>>> qe.d_cell(Integer(3), Integer(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.
<BLANKLINE>
alpha = the unique root of x^2 - 3 between 0 and 4
      = 1.7320508076-
<BLANKLINE>
Coordinate 1 = 0
             = 0.0000000000
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)
```

```
>>> from sage.all import *
>>> c = qe.cell(Integer(3), Integer(4)); c                     # optional - qepcad
QEPCAD cell (3, 4)
```

A *QepcadCell* has accessor methods for the important state held within a cell. For instance:

```
sage: # optional - qepcad
sage: c.level()
2
sage: c.index()
```

(continues on next page)

(continued from previous page)

```
(3, 4)
sage: qe.cell(3).number_of_children()
5
sage: len(qe.cell(3))
5
```

```
>>> from sage.all import *
>>> # optional - qepcad
>>> c.level()
2
>>> c.index()
(3, 4)
>>> qe.cell(Integer(3)).number_of_children()
5
>>> len(qe.cell(Integer(3)))
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?}
```

```
>>> from sage.all import *
>>> c.sample_point()                          # optional - qepcad
(0, 1.732050807568878?)
>>> 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)]
```

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

```
>>> from sage.all import *
>>> c = qe.cell(Integer(3))                              # optional - qepcad
>>> c[Integer(1)]                                        # optional - qepcad
QEPCAD cell (3, 1)
>>> [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]
```

```
>>> from sage.all import *
>>> F = qf.exactly_k(Integer(2), y, circle == Integer(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]
```

```
>>> from sage.all import *
>>> G = qf.exactly_k(Integer(2), y, ellipse == Integer(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
```

```
>>> from sage.all import *
>>> 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␣
↪- qepcad
```

(continues on next page)

(continued from previous page)

```
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'
```

```
>>> from sage.all import *
>>> qe = qepcad(qf.and_(ellipse == Integer(0), circle == Integer(0)), interact=True) ␣
↪     # optional - qepcad
>>> 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
```

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

```
>>> from sage.all import *
>>> qe.cell(Integer(12),Integer(2)).signs()[Integer(1)][Integer(0)]          ␣
↪    # optional - qepcad
```

(continues on next page)

(continued from previous page)

```
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:
....:         count_ellipse += (c2.signs()[1][0] == 0)
....:         count_circle += (c2.signs()[1][1] == 0)
....:     c.set_truth(count_ellipse == 2 and count_circle == 2)
```

```
>>> from sage.all import *
>>> for c in qe.cell():                              # optional - qepcad
...     count_ellipse = Integer(0)
...     count_circle = Integer(0)
...     for c2 in c:
...         count_ellipse += (c2.signs()[Integer(1)][Integer(0)] == Integer(0))
...         count_circle += (c2.signs()[Integer(1)][Integer(1)] == Integer(0))
...     c.set_truth(count_ellipse == Integer(2) and count_circle == Integer(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
```

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

```
>>> from sage.all import *
>>> qe = qepcad(x**Integer(3) - x == Integer(0), interact=True)   # optional -␣
↪qepcad
>>> qe.finish()                                 # not tested (random order)
x - 1 <= 0 /\ x + 1 >= 0 /\ [ x = 0 \/ x - 1 = 0 \/ x + 1 = 0 ]
>>> 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 >= 0. Without the assumption we'd get something like [a = 0 /b /= 0] / [a /= 0 /4 a c - b^2 <= 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
sage: qe.assume(a != 0) # optional - qepcad
sage: qe.finish() # optional - qepcad
4 a c - b^2 <= 0
```

```
>>> from sage.all import *
>>> var('a,b,c,x')
(a, b, c, x)
>>> qf = qepcad_formula
>>> qe = qepcad(qf.exists(x, a*x**Integer(2) + b*x + c == Integer(0)),␣
↪interact=True) # optional - qepcad
>>> qe.assume(a != Integer(0)) # optional - qepcad
>>> 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: # optional - qepcad
sage: qe = qepcad(x + 3 == 42, interact=True)
sage: qe.go(); qe.go(); qe.go()
```

(continues on next page)

(continued from previous page)

```
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)
QEPCAD cell (2)
sage: qe.cell(2) is qe.cell(2)
True
```

```
>>> from sage.all import *
>>> # optional - qepcad
>>> qe = qepcad(x + Integer(3) == Integer(42), interact=True)
>>> qe.go(); qe.go(); qe.go()
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'
>>> qe.cell(Integer(2))
QEPCAD cell (2)
>>> qe.cell(Integer(2)) is qe.cell(Integer(2))
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.
-----------------------------------------------------------------------------
```

```
>>> from sage.all import *
>>> qe = qepcad(x == Integer(0), interact=True)  # optional - qepcad
>>> qe.finish()  # optional - qepcad
x = 0
>>> 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)]
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> qe = qepcad(x**Integer(2) + y**Integer(2) == Integer(1), interact=True) #␣
↪optional - qepcad
>>> 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'
>>> 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: # optional - qepcad
sage: qe = qepcad(x > 2/3, interact=True)
sage: qe.phase()
'Before Normalization'
sage: qe.go()
QEPCAD object has moved to phase 'At the end of projection phase'
sage: qe.phase()
'At the end of projection phase'
sage: qe.go()
QEPCAD object has moved to phase 'Before Choice'
sage: qe.phase()
'Before Choice'
sage: qe.go()
QEPCAD object has moved to phase 'Before Solution'
sage: qe.phase()
'Before Solution'
sage: qe.go()
3 x - 2 > 0
sage: qe.phase()
'EXITED'
```

```
>>> from sage.all import *
>>> # optional - qepcad
>>> qe = qepcad(x > Integer(2)/Integer(3), interact=True)
>>> qe.phase()
'Before Normalization'
>>> qe.go()
QEPCAD object has moved to phase 'At the end of projection phase'
>>> qe.phase()
'At the end of projection phase'
>>> qe.go()
QEPCAD object has moved to phase 'Before Choice'
>>> qe.phase()
'Before Choice'
>>> qe.go()
QEPCAD object has moved to phase 'Before Solution'
>>> qe.phase()
'Before Solution'
>>> qe.go()
3 x - 2 > 0
>>> qe.phase()
'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
```

```
>>> from sage.all import *
>>> qe = qepcad(x == Integer(1), interact=True) # optional - qepcad
>>> 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'
>>> qe.set_truth_value(Integer(1), Integer(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 ]
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
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> qf = qepcad_formula
>>> qe = qepcad(qf.and_(x**Integer(2) + y**Integer(2) - Integer(3) ==␣
↪Integer(0), x + y > Integer(0)), interact=True) # optional - qepcad
>>> 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'
>>> 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 ]
>>> qe.solution_extension('G')                    # not tested (random order)
[
  [
```

(continues on next page)

(continued from previous page)

```
    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
]
>>> 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)
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> qe = qepcad(x**Integer(2) + y**Integer(2) == Integer(1), interact=True) #␣
↪optional - qepcad
>>> 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'
>>> qe.cell().index() # optional - qepcad
()
```

(continues on next page)

(continued from previous page)

```
>>> qe.cell(Integer(1)).index() # optional - qepcad
(1,)
>>> qe.cell(Integer(2), Integer(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
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
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> qe = qepcad(x**Integer(2) + y**Integer(2) == Integer(1), interact=True) #␣
↪optional - qepcad
>>> 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'
>>> qe.cell().level() # optional - qepcad
0
>>> qe.cell(Integer(1)).level() # optional - qepcad
1
>>> qe.cell(Integer(2), Integer(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'
```

(continues on next page)

(continued from previous page)

```
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]
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> qe = qepcad(x**Integer(2) + y**Integer(2) == Integer(1), interact=True) #␣
↪optional - qepcad
>>> 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'
>>> qe.cell().number_of_children() # optional - qepcad
5
>>> [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: # optional - qepcad
sage: qe = qepcad(x^2 - x - 1 == 0, interact=True)
sage: qe.go(); qe.go(); qe.go()
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
-0.618033988749895?
sage: v2 = qe.cell(4).sample_point()[0]; v2
1.618033988749895?
sage: v1 + v2 == 1
True
```

```
>>> from sage.all import *
>>> # optional - qepcad
>>> qe = qepcad(x**Integer(2) - x - Integer(1) == Integer(0), interact=True)
>>> qe.go(); qe.go(); qe.go()
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'
>>> v1 = qe.cell(Integer(2)).sample_point()[Integer(0)]; v1
-0.618033988749895?
>>> v2 = qe.cell(Integer(4)).sample_point()[Integer(0)]; v2
1.618033988749895?
>>> v1 + v2 == Integer(1)
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'
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?}
```

```
>>> from sage.all import *
>>> qe = qepcad(x**Integer(2) - x - Integer(1) == Integer(0), interact=True)
↪# optional - qepcad
>>> 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'
>>> qe.cell(Integer(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
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> qe = qepcad(x**Integer(2) + y**Integer(2) == Integer(1), interact=True) #␣
↪optional - qepcad
>>> 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'
>>> qe.solution_extension('T') # optional - qepcad
y^2 + x^2 - 1 = 0
>>> qe.cell(Integer(3), Integer(3)).set_truth(True) # optional - qepcad
```

(continues on next page)

(continued from previous page)

```
>>> 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
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)]
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> qe = qepcad(x**Integer(2) + y**Integer(2) == Integer(1), interact=True) #␣
↪optional - qepcad
>>> 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'
>>> from sage.interfaces.qepcad import QepcadCell
>>> all_cells = flatten(qe.cell(), ltypes=QepcadCell, max_level=Integer(1)) #␣
↪optional - qepcad
>>> [(c, c.signs()[Integer(1)][Integer(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: *ExpectFunction*

A wrapper for a QEPCAD command.

class sage.interfaces.qepcad.**Qepcad_expect**(*memcells=None*, *maxread=None*, *logfile=None*, *server=None*)

Bases: `ExtraTabCompletion`, *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
```

```
>>> from sage.all import *
>>> qf = qepcad_formula

>>> var('a,b,c,d,x,y,z,long_with_underscore_314159')
(a, b, c, d, x, y, z, long_with_underscore_314159)
>>> K = QQ['q, r']; (q, r,) = K._first_ngens(2)
```

(continues on next page)

(continued from previous page)

```
>>> qepcad('(E x)[a x + b > 0]', vars='(a,b,x)')       # not tested (random order)
a /= 0 \/ b > 0

>>> qepcad(a > b)                               # optional - qepcad
b - a < 0

>>> qepcad(qf.exists(x, a*x**Integer(2) + b*x + c == Integer(0)))        # not␣
↪tested (random order)
4 a c - b^2 <= 0 /\ [ c = 0 \/ a /= 0 \/ 4 a c - b^2 < 0 ]

>>> qepcad(qf.exists(x, a*x**Integer(2) + b*x + c == Integer(0)), assume=(a !=␣
↪Integer(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
```

```
>>> from sage.all import *
>>> exact2 = qepcad(qf.exactly_k(Integer(2), x, a*x**Integer(2) + b*x + c ==␣
↪Integer(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 ]
```

```
>>> from sage.all import *
>>> exact1 = qepcad(qf.exactly_k(Integer(1), x, a*x**Integer(2) + b*x + c ==␣
↪Integer(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 ]
```

```
>>> from sage.all import *
>>> exact0 = qepcad(qf.forall(x, a*x**Integer(2) + b*x + c != Integer(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
```

```
>>> from sage.all import *
>>> qepcad(qf.exactly_k(Integer(3)**Integer(75), x, a*x**Integer(2) + b*x + c ==␣
↪Integer(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
```

```
>>> from sage.all import *
>>> qepcad(r'[[%s] /\ [%s]]' % (exact0, exact1), vars='a,b,c')      # not tested␣
↪(random order)
FALSE
>>> qepcad(r'[[%s] /\ [%s]]' % (exact0, exact2), vars='a,b,c')      # not tested␣
↪(random order)
FALSE
>>> 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
```

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

```
>>> from sage.all import *
>>> qepcad(qf.infinitely_many(x, a*x**Integer(2) + b*x + c == Integer(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
```

```
>>> from sage.all import *
>>> qepcad(qf.all_but_finitely_many(x, a*x**Integer(2) + b*x + c != Integer(0))) ␣
↪  # not tested (random order)
b /= 0 \/ a /= 0 \/ c /= 0
```

The nonzeroes 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 ]
```

```
>>> from sage.all import *
>>> qepcad(qf.connected_subset(x, a*x**Integer(2) + b*x + c != Integer(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
```

```
>>> from sage.all import *
>>> qepcad(qf.connected_subset(x, a*x**Integer(2) + b*x + c == Integer(0)))     ␣
↪  # not tested (random order)
a = 0 \/ 4 a c - b^2 >= 0
>>> 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
```

```
>>> from sage.all import *
>>> qepcad(qf.infinitely_many(x, a*x**Integer(2) + b*x + c > Integer(0)))
↪# not tested (random order)
c > 0 \/ a > 0 \/ 4 a c - b^2 < 0
```

(continues on next page)

(continued from previous page)

```
>>> qepcad(qf.exists(x, a*x**Integer(2) + b*x + c > Integer(0)))
↪# not tested (random order)
c > 0 \/ a > 0 \/ 4 a c - b^2 < 0
```

However, since $y >= 0$ 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
```

```
>>> from sage.all import *
>>> qepcad(qf.infinitely_many(x, a*x**Integer(2) + b*x + c >= Integer(0)),␣
↪assume=(a != Integer(0)))     # not tested (random order)
a > 0 \/ 4 a c - b^2 < 0
>>> qepcad(qf.exists(x, a*x**Integer(2) + b*x + c >= Integer(0)), assume=(a !=␣
↪Integer(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
=======================================================
```

```
>>> from sage.all import *
>>> from sage.interfaces.qepcad import qepcad_banner
>>> qepcad_banner() # optional - qepcad
=======================================================
                Quantifier Elimination
                          in
            Elementary Algebra and Geometry
```

(continues on next page)

(continued from previous page)

```
                          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 ']':
```

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

```
>>> from sage.all import *
>>> var('a,b')
```

(continues on next page)

(continued from previous page)

```
(a, b)
>>> qf = qepcad_formula
>>> qf.forall(a, a**Integer(2) + b > b**Integer(2) + a)
(A a)[a^2 + b > b^2 + a]
>>> qf.forall((a, b), a**Integer(2) + b**Integer(2) > Integer(0))
(A a)(A b)[a^2 + b^2 > 0]
>>> qf.A(b, b**Integer(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]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.connected_subset(a, a**Integer(2) + b > b**Integer(2) + a)
(C a)[a^2 + b > b^2 + a]
>>> qf.C(b, b**Integer(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]
```

(continues on next page)

(continued from previous page)

```
sage: qf.E(b, b^2 == a)
(E b)[b^2 = a]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.exists(a, a**Integer(2) + b > b**Integer(2) + a)
(E a)[a^2 + b > b^2 + a]
>>> qf.exists((a, b), a**Integer(2) + b**Integer(2) < Integer(0))
(E a)(E b)[a^2 + b^2 < 0]
>>> qf.E(b, b**Integer(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]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.infinitely_many(a, a**Integer(2) + b > b**Integer(2) + a)
(F a)[a^2 + b > b^2 + a]
>>> qf.F(b, b**Integer(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]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.all_but_finitely_many(a, a**Integer(2) + b > b**Integer(2) + a)
(G a)[a^2 + b > b^2 + a]
>>> qf.G(b, b**Integer(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]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.exactly_k(Integer(1), x, x**Integer(2) + a*x + b == Integer(0))
(X1 x)[a x + x^2 + b = 0]
>>> qf.exactly_k(Integer(0), b, a*b == Integer(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]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.all_but_finitely_many(a, a**Integer(2) + b > b**Integer(2) + a)
(G a)[a^2 + b > b^2 + a]
>>> qf.G(b, b**Integer(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]]
```

```
>>> from sage.all import *
>>> var('a,b,c,x')
(a, b, c, x)
>>> qf = qepcad_formula
>>> qf.and_(a*b, a*c, b*c != Integer(0))
[a b = 0 /\ a c = 0 /\ b c /= 0]
>>> qf.and_(a*x**Integer(2) == Integer(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'}))
```

```
>>> from sage.all import *
>>> qf = qepcad_formula
>>> var('a,b,c')
(a, b, c)
>>> K = QQ['x, y']; (x, y,) = K._first_ngens(2)
>>> def test_qf(qf):
...     return qf, qf.vars
>>> test_qf(qf.atomic(a**Integer(2) + Integer(17)))
(a^2 + 17 = 0, frozenset({'a'}))
>>> test_qf(qf.atomic(a*b*c <= c**Integer(3)))
(a b c <= c^3, frozenset({'a', 'b', 'c'}))
>>> test_qf(qf.atomic(x+y**Integer(2), '!=', a+b))
(y^2 + x /= a + b, frozenset({'a', 'b', 'x', 'y'}))
>>> 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]
```

(continues on next page)

(continued from previous page)

```
sage: qf.C(b, b^2 != a)
(C b)[b^2 /= a]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.connected_subset(a, a**Integer(2) + b > b**Integer(2) + a)
(C a)[a^2 + b > b^2 + a]
>>> qf.C(b, b**Integer(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]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.exactly_k(Integer(1), x, x**Integer(2) + a*x + b == Integer(0))
(X1 x)[a x + x^2 + b = 0]
>>> qf.exactly_k(Integer(0), b, a*b == Integer(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]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.exists(a, a**Integer(2) + b > b**Integer(2) + a)
(E a)[a^2 + b > b^2 + a]
>>> qf.exists((a, b), a**Integer(2) + b**Integer(2) < Integer(0))
(E a)(E b)[a^2 + b^2 < 0]
>>> qf.E(b, b**Integer(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]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.forall(a, a**Integer(2) + b > b**Integer(2) + a)
(A a)[a^2 + b > b^2 + a]
>>> qf.forall((a, b), a**Integer(2) + b**Integer(2) > Integer(0))
(A a)(A b)[a^2 + b^2 > 0]
>>> qf.A(b, b**Integer(2) != a)
(A b)[b^2 /= a]
```

formula(*formula*)

Construct 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]
```

```
>>> from sage.all import *
>>> var('a,b,c,x')
(a, b, c, x)
>>> qf = qepcad_formula
>>> qf.formula(a*x + b)
a x + b = 0
>>> qf.formula((a*x**Integer(2) + b*x + c, a != Integer(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]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.iff(a, b)
[a = 0 <==> b = 0]
>>> qf.iff(a**Integer(2) < b, b**Integer(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]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.implies(a, b)
[a = 0 ==> b = 0]
>>> qf.implies(a**Integer(2) < b, b**Integer(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)
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]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.infinitely_many(a, a**Integer(2) + b > b**Integer(2) + a)
(F a)[a^2 + b > b^2 + a]
>>> qf.F(b, b**Integer(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]]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.not_(a > b)
[~a > b]
>>> qf.not_(a**Integer(2) + b**Integer(2))
[~a^2 + b^2 = 0]
>>> qf.not_(qf.and_(a > Integer(0), b < Integer(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]]
```

```
>>> from sage.all import *
>>> var('a,b,c,x')
(a, b, c, x)
>>> qf = qepcad_formula
>>> qf.or_(a*b, a*c, b*c != Integer(0))
[a b = 0 \/ a c = 0 \/ b c /= 0]
>>> qf.or_(a*x**Integer(2) == Integer(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)
(NOT_A_REAL_QEPCAD_QUANTIFIER a)[a b = 0]
sage: qf.quantifier('FOO', (a, b), a*b)
(FOO a)(FOO b)[a b = 0]
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> qf = qepcad_formula
>>> qf.quantifier('NOT_A_REAL_QEPCAD_QUANTIFIER', a, a*b==Integer(0))
(NOT_A_REAL_QEPCAD_QUANTIFIER a)[a b = 0]
>>> 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'
```

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

CHAPTER

FORTY

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 decribed in the Notebook manual.

The %R and %%R interface creating an R line or an R cell in the Jupyter notebook are briefly decribed 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
[1] 10.4  5.6  3.1  6.4 21.7
```

```
>>> from sage.all import *
>>> x = r([RealNumber('10.4'),RealNumber('5.6'),RealNumber('3.1'),RealNumber('6.4'),
↪RealNumber('21.7')]); x
[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
[1] 0.09615385 0.17857143 0.32258065 0.15625000 0.04608295
sage: 1/x
[1] 0.09615385 0.17857143 0.32258065 0.15625000 0.04608295
```

```
>>> from sage.all import *
>>> ~x
[1] 0.09615385 0.17857143 0.32258065 0.15625000 0.04608295
>>> Integer(1)/x
[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
[1] 10.4  5.6  3.1  6.4 21.7  0.0 10.4  5.6  3.1  6.4 21.7
```

```
>>> from sage.all import *
>>> y = r([x,Integer(0),x]); y
[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
[1] 32.2 17.8 10.3 20.2 66.1 21.8 22.6 12.8 16.9 50.8 43.5
```

```
>>> from sage.all import *
>>> v = Integer(2)*x+y+Integer(1); v
[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)
[1] 47.2
sage: x.sum()
[1] 47.2
```

```
>>> from sage.all import *
>>> sum(x)
[1] 47.2
>>> x.sum()
[1] 47.2
```

One can calculate the sample variance of a list of numbers:

```
sage: ((x-x.mean())^2/(x.length()-1)).sum()
[1] 53.853
sage: x.var()
[1] 53.853

sage: x.sort()
[1] 3.1  5.6  6.4 10.4 21.7
sage: x.min()
[1] 3.1
sage: x.max()
[1] 21.7
sage: x
[1] 10.4  5.6  3.1  6.4 21.7

sage: r(-17).sqrt()
[1] NaN
sage: r('-17+0i').sqrt()
[1] 0+4.123106i
```

```
>>> from sage.all import *
>>> ((x-x.mean())**Integer(2)/(x.length()-Integer(1))).sum()
[1] 53.853
```

(continues on next page)

(continued from previous page)

```
>>> x.var()
[1] 53.853

>>> x.sort()
[1] 3.1  5.6  6.4 10.4 21.7
>>> x.min()
[1] 3.1
>>> x.max()
[1] 21.7
>>> x
[1] 10.4  5.6  3.1  6.4 21.7

>>> r(-Integer(17)).sqrt()
[1] NaN
>>> r('-17+0i').sqrt()
[1] 0+4.123106i
```

Generating an arithmetic sequence:

```
sage: r('1:10')
[1] 1  2  3  4  5  6  7  8  9 10
```

```
>>> from sage.all import *
>>> r('1:10')
[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)
[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])
sage: x.rep(2)
[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)
[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)
[1] 10.4 10.4  5.6  5.6  3.1  3.1  6.4  6.4 21.7 21.7
```

```
>>> from sage.all import *
>>> r.seq(length=Integer(10), from_=-Integer(1), by=RealNumber('.2'))
[1] -1.0 -0.8 -0.6 -0.4 -0.2  0.0  0.2  0.4  0.6  0.8

>>> x = r([RealNumber('10.4'),RealNumber('5.6'),RealNumber('3.1'),RealNumber('6.4'),
↪RealNumber('21.7')])
>>> x.rep(Integer(2))
[1] 10.4  5.6  3.1  6.4 21.7 10.4  5.6  3.1  6.4 21.7
>>> x.rep(times=Integer(2))
[1] 10.4  5.6  3.1  6.4 21.7 10.4  5.6  3.1  6.4 21.7
>>> x.rep(each=Integer(2))
[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')
sage: z = r([1,2,3,na])
sage: z
[1]  1  2  3 NA
sage: ind = r.is_na(z)
sage: ind
[1] FALSE FALSE FALSE  TRUE
sage: zero = r(0)
sage: zero / zero
[1] NaN
sage: inf = r('Inf')
sage: inf-inf
[1] NaN
sage: r.is_na(inf)
[1] FALSE
sage: r.is_na(inf-inf)
[1] TRUE
sage: r.is_na(zero/zero)
[1] TRUE
sage: r.is_na(na)
[1] TRUE
sage: r.is_nan(inf-inf)
[1] TRUE
sage: r.is_nan(zero/zero)
[1] TRUE
sage: r.is_nan(na)
[1] FALSE
```

```
>>> from sage.all import *
>>> na = r('NA')
>>> z = r([Integer(1),Integer(2),Integer(3),na])
>>> z
[1]  1  2  3 NA
>>> ind = r.is_na(z)
>>> ind
[1] FALSE FALSE FALSE  TRUE
>>> zero = r(Integer(0))
>>> zero / zero
[1] NaN
>>> inf = r('Inf')
>>> inf-inf
[1] NaN
>>> r.is_na(inf)
[1] FALSE
>>> r.is_na(inf-inf)
[1] TRUE
>>> r.is_na(zero/zero)
[1] TRUE
>>> r.is_na(na)
[1] TRUE
>>> r.is_nan(inf-inf)
```

(continues on next page)

(continued from previous page)

```
[1] TRUE
>>> r.is_nan(zero/zero)
[1] TRUE
>>> r.is_nan(na)
[1] FALSE
```

Character Vectors:

```
sage: labs = r.paste('c("X","Y")', '1:10', sep='""'); labs
[1] "X1"  "Y2"  "X3"  "Y4"  "X5"  "Y6"  "X7"  "Y8"  "X9"  "Y10"
```

```
>>> from sage.all import *
>>> labs = r.paste('c("X","Y")', '1:10', sep='""'); labs
[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')
sage: x = r([10.4,5.6,3.1,6.4,21.7,na]); x
[1] 10.4  5.6  3.1  6.4 21.7   NA
sage: x['!is.na(self)']
[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
[1] 10.4  5.6  3.1  6.4 21.7   NA
sage: (x+1)['(!is.na(self)) & self>0']
[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
[1] 10.4 -2.0  3.1 -0.5 21.7   NA
sage: (x+1)['(!is.na(self)) & self>0']
[1] 11.4  4.1  0.5 22.7
```

```
>>> from sage.all import *
>>> na = r('NA')
>>> x = r([RealNumber('10.4'),RealNumber('5.6'),RealNumber('3.1'),RealNumber('6.4'),
↪RealNumber('21.7'),na]); x
[1] 10.4  5.6  3.1  6.4 21.7   NA
>>> x['!is.na(self)']
[1] 10.4  5.6  3.1  6.4 21.7

>>> x = r([RealNumber('10.4'),RealNumber('5.6'),RealNumber('3.1'),RealNumber('6.4'),
↪RealNumber('21.7'),na]); x
[1] 10.4  5.6  3.1  6.4 21.7   NA
>>> (x+Integer(1))['(!is.na(self)) & self>0']
[1] 11.4  6.6  4.1  7.4 22.7
>>> x = r([RealNumber('10.4'),-Integer(2),RealNumber('3.1'),-RealNumber('0.5'),
↪RealNumber('21.7'),na]); x
[1] 10.4 -2.0  3.1 -0.5 21.7   NA
>>> (x+Integer(1))['(!is.na(self)) & self>0']
[1] 11.4  4.1  0.5 22.7
```

Distributions:

```
sage: r.options(width='60')
$width
[1] 80

sage: rr = r.dnorm(r.seq(-3,3,0.1))
sage: rr
 [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
[13] 0.078950158 0.094049077 0.110920835 0.129517596
[17] 0.149727466 0.171368592 0.194186055 0.217852177
[21] 0.241970725 0.266085250 0.289691553 0.312253933
[25] 0.333224603 0.352065327 0.368270140 0.381387815
[29] 0.391042694 0.396952547 0.398942280 0.396952547
[33] 0.391042694 0.381387815 0.368270140 0.352065327
[37] 0.333224603 0.312253933 0.289691553 0.266085250
[41] 0.241970725 0.217852177 0.194186055 0.171368592
[45] 0.149727466 0.129517596 0.110920835 0.094049077
[49] 0.078950158 0.065615815 0.053990967 0.043983596
[53] 0.035474593 0.028327038 0.022394530 0.017528300
[57] 0.013582969 0.010420935 0.007915452 0.005952532
[61] 0.004431848
```

```
>>> from sage.all import *
>>> r.options(width='60')
$width
[1] 80

>>> rr = r.dnorm(r.seq(-Integer(3),Integer(3),RealNumber('0.1')))
>>> rr
 [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
[13] 0.078950158 0.094049077 0.110920835 0.129517596
[17] 0.149727466 0.171368592 0.194186055 0.217852177
[21] 0.241970725 0.266085250 0.289691553 0.312253933
[25] 0.333224603 0.352065327 0.368270140 0.381387815
[29] 0.391042694 0.396952547 0.398942280 0.396952547
[33] 0.391042694 0.381387815 0.368270140 0.352065327
[37] 0.333224603 0.312253933 0.289691553 0.266085250
[41] 0.241970725 0.217852177 0.194186055 0.171368592
[45] 0.149727466 0.129517596 0.110920835 0.094049077
[49] 0.078950158 0.065615815 0.053990967 0.043983596
[53] 0.035474593 0.028327038 0.022394530 0.017528300
[57] 0.013582969 0.010420935 0.007915452 0.005952532
[61] 0.004431848
```

Convert R Data Structures to Python/Sage:

```
sage: rr = r.dnorm(r.seq(-3,3,0.1))
sage: sum(rr._sage_())
9.9772125168981...
```

```
>>> from sage.all import *
>>> rr = r.dnorm(r.seq(-Integer(3),Integer(3),RealNumber('0.1')))
>>> sum(rr._sage_())
9.9772125168981...
```

Or you get a dictionary to be able to access all the information:

```
sage: rs = r.summary(r.c(1,4,3,4,3,2,5,1))
sage: rs
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   1.750   3.000   2.875   4.000   5.000
  sage: d = rs._sage_()
  sage: d['DATA']
  [1, 1.75, 3, 2.875, 4, 5]
  sage: d['_Names']
  ['Min.', '1st Qu.', 'Median', 'Mean', '3rd Qu.', 'Max.']
  sage: d['_r_class']
  ['summaryDefault', 'table']
```

```
>>> from sage.all import *
>>> rs = r.summary(r.c(Integer(1),Integer(4),Integer(3),Integer(4),Integer(3),
↪Integer(2),Integer(5),Integer(1)))
>>> rs
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   1.750   3.000   2.875   4.000   5.000
>>> d = rs._sage_()
>>> d['DATA']
  [1, 1.75, 3, 2.875, 4, 5]
>>> d['_Names']
  ['Min.', '1st Qu.', 'Median', 'Mean', '3rd Qu.', 'Max.']
>>> d['_r_class']
  ['summaryDefault', 'table']
```

It is also possible to access the plotting capabilities of R through Sage. For more information see the documentation of r.plot() or r.png().

THE JUPYTER NOTEBOOK INTERFACE (work in progress).

The %r interface described in the Sage notebook manual is not useful in the Jupyter notebook : it creates a inferior R interpreter which cannot be escaped.

The RPy2 library allows the creation of an R cell in the Jupyter notebook analogous to the %r escape in command line or %r cell in a Sage notebook.

The interface is loaded by a cell containing the sole code:

“%load_ext rpy2.ipython”

After execution of this code, the %R and %%R magics are available:

- **%R allows the execution of a single line of R code. Data exchange is**
  possible via the -i and -o options. Do “%R?” in a standalone cell to get the documentation.
- **%%R allows the execution in R of the whole text of a cell, with**
  similar options (do “%%R?” in a standalone cell for documentation).

A few important points must be noted:

- The R interpreter launched by this interface IS (currently) DIFFERENT from the R interpreter used br 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: `ExtraTabCompletion`, *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))
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
1.00    2.00    3.00   12.55    3.50  111.00
```

```
>>> from sage.all import *
>>> r.summary(r.c(Integer(1),Integer(2),Integer(3),Integer(111),Integer(2),
↪Integer(3),Integer(2),Integer(3),Integer(2),Integer(5),Integer(4)))
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
1.00    2.00    3.00   12.55    3.50  111.00
```

available_packages()

Return 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 is that is checked is defined at the top of sage/interfaces/r.py.

EXAMPLES:

```
sage: ap = r.available_packages()                    # optional - internet
sage: len(ap) > 20                                   # optional - internet
True
```

```
>>> from sage.all import *
>>> ap = r.available_packages()                    # optional - internet
>>> len(ap) > Integer(20)                                   # optional -␣
↪internet
True
```

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

This is an alias for *function_call()*.

EXAMPLES:

```
sage: r.call('length', [1,2,3])
[1] 3
```

```
>>> from sage.all import *
>>> r.call('length', [Integer(1),Integer(2),Integer(3)])
[1] 3
```

chdir(*dir*)

Changes the working directory to `dir`.

INPUT:

- `dir` – the directory to change to

EXAMPLES:

```
sage: import tempfile
sage: tmpdir = tempfile.mkdtemp()
sage: r.chdir(tmpdir)
```

```
>>> from sage.all import *
>>> import tempfile
>>> tmpdir = tempfile.mkdtemp()
>>> r.chdir(tmpdir)
```

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 Issue #10264).

```
sage: os.path.realpath(tmpdir) == sageobj(r.getwd())  # known bug (issue
↪#9970)
True
```

```
>>> from sage.all import *
>>> os.path.realpath(tmpdir) == sageobj(r.getwd())  # known bug (issue #9970)
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 of strings

EXAMPLES:

```
sage: dummy = r._tab_completion(use_disk_cache=False)  # clean doctest
sage: 'testInheritedMethods' in r.completions('tes')
True
```

```
>>> from sage.all import *
>>> dummy = r._tab_completion(use_disk_cache=False)  # clean doctest
>>> 'testInheritedMethods' in r.completions('tes')
True
```

console()

Run the R console as a separate new R process.

EXAMPLES:

```
sage: r.console()                    # not tested
    R version 2.6.1 (2007-11-26)
    Copyright (C) 2007 The R Foundation for Statistical Computing
    ISBN 3-900051-07-0
    ...
```

```
>>> from sage.all import *
>>> r.console()                    # not tested
    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*)

Convert 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")'
sage: r.convert_r_list(s)
['.GlobalEnv',
 'package:stats',
 'package:graphics',
```

(continues on next page)

(continued from previous page)

```
 'package:grDevices',
 'package:utils',
 'package:datasets',
 'package:methods',
 'Autoloads',
 'package:base']
```

```
>>> from sage.all import *
>>> s = 'c(".GlobalEnv", "package:stats", "package:graphics",
↪"package:grDevices", \n"package:utils", "package:datasets", "package:methods
↪", "Autoloads", \n"package:base")'
>>> r.convert_r_list(s)
['.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')
'[1] 2'
```

```
>>> from sage.all import *
>>> r.eval('1+1')
'[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] ])
[1] 3
```

```
>>> from sage.all import *
>>> r.function_call('length', args=[ [Integer(1),Integer(2),Integer(3)] ])
[1] 3
```

get(*var*)

Return the string representation of the variable var.

INPUT:

- `var` – string

OUTPUT: string

EXAMPLES:

```
sage: r.set('a', 2)
sage: r.get('a')
'[1] 2'
```

```
>>> from sage.all import *
>>> r.set('a', Integer(2))
>>> r.get('a')
'[1] 2'
```

help(*command*)

Return help string for a given command.

INPUT:

- `command` – string

OUTPUT: HelpExpression; a subclass of string whose `__repr__` method is `__str__`, so it prints nicely

EXAMPLES:

```
sage: r.help('c')
title
-----

Combine Values into a Vector or List

name
----

c
...
```

```
>>> from sage.all import *
>>> r.help('c')
title
-----
<BLANKLINE>
Combine Values into a Vector or List
<BLANKLINE>
name
----
<BLANKLINE>
c
...
```

install_packages(*package_name*)

Install an R package into Sage's R installation.

EXAMPLES:

```
sage: r.install_packages('aaMI')       # not tested
...
```

(continues on next page)

(continued from previous page)

```
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'.
```

```
>>> from sage.all import *
>>> r.install_packages('aaMI')       # not tested
...
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')
sage: 'grid' in r.eval('(.packages())')
True
sage: r.library('foobar')
Traceback (most recent call last):
...
ImportError: ...
```

```
>>> from sage.all import *
>>> r.library('grid')
>>> 'grid' in r.eval('(.packages())')
True
>>> r.library('foobar')
Traceback (most recent call last):
...
ImportError: ...
```

na ()

Return the NA in R.

OUTPUT: RElement; an element of R

EXAMPLES:

```
sage: r.na()
[1] NA
```

```
>>> from sage.all import *
>>> r.na()
[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 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 - rgraphics
....:     _ = r.setwd(d)
....:     r.plot("1:10")
null device
          1
```

```
>>> from sage.all import *
>>> from tempfile import TemporaryDirectory
>>> with TemporaryDirectory() as d:                # optional - 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'
sage: r.png(filename='"%s"'%filename)                # optional - rgraphics
NULL
sage: x = r([1,2,3])
sage: y = r([4,5,6])
sage: r.plot(x,y)                                    # optional - rgraphics
null device
          1
sage: import os; os.unlink(filename)                 # optional - rgraphics
```

```
>>> from sage.all import *
>>> filename = tmp_filename() + '.png'
>>> r.png(filename='"%s"'%filename)                # optional - rgraphics
NULL
>>> x = r([Integer(1),Integer(2),Integer(3)])
>>> y = r([Integer(4),Integer(5),Integer(6)])
```

(continues on next page)

(continued from previous page)

```
>>> r.plot(x,y)                                        # optional - rgraphics
null device
          1
>>> import os; os.unlink(filename)                     # optional - rgraphics
```

Please note that for more extensive use of R's plotting capabilities (such as the lattices 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: # not tested
sage: r.X11()
sage: r.quartz()
sage: r.hist("rnorm(100)")
sage: r.library("lattice")
sage: r.histogram(x = '~ wt | cyl', data='mtcars')
sage: r.dev_off()
```

```
>>> from sage.all import *
>>> # not tested
>>> r.X11()
>>> r.quartz()
>>> r.hist("rnorm(100)")
>>> r.library("lattice")
>>> r.histogram(x = '~ wt | cyl', data='mtcars')
>>> r.dev_off()
```

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
sage: r.png(filename='"%s"'%filename)              # optional - rgraphics
NULL
sage: r.library("lattice")
sage: r("print(histogram(~wt | cyl, data=mtcars))")        # optional -␣
↪rgraphics
sage: import os; os.unlink(filename)               # optional - rgraphics
```

```
>>> from sage.all import *
>>> filename = tmp_filename() + '.png'  # not needed in notebook, used for␣
↪doctesting
>>> r.png(filename='"%s"'%filename)              # optional - rgraphics
NULL
>>> r.library("lattice")
>>> r("print(histogram(~wt | cyl, data=mtcars))")        # optional -␣
↪rgraphics
>>> import os; os.unlink(filename)               # optional - rgraphics
```

png(**args*, ***kwds*)

Create 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'
sage: r.png(filename='"%s"'%filename)              # optional - rgraphics
NULL
sage: x = r([1,2,3])
sage: y = r([4,5,6])
sage: r.plot(x,y)                                  # optional - rgraphics
null device
          1
sage: import os; os.unlink(filename)               # optional - rgraphics
```

```
>>> from sage.all import *
>>> filename = tmp_filename() + '.png'
>>> r.png(filename='"%s"'%filename)                # optional - rgraphics
NULL
>>> x = r([Integer(1),Integer(2),Integer(3)])
>>> y = r([Integer(4),Integer(5),Integer(6)])
>>> r.plot(x,y)                                    # optional - rgraphics
null device
          1
>>> import os; os.unlink(filename)                 # optional - rgraphics
```

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
sage: t = r.eval('capabilities("aqua")')   # should be on all supported Mac␣
↪versions
sage: "TRUE" in s+t                                # optional - rgraphics
True
```

```
>>> from sage.all import *
>>> s = r.eval('capabilities("png")')    # should be on Linux and Solaris
>>> t = r.eval('capabilities("aqua")')   # should be on all supported Mac␣
↪versions
>>> "TRUE" in s+t                                  # optional - rgraphics
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()
sage: f = open(filename, 'w')
sage: _ = f.write('a <- 2+2\n')
sage: f.close()
sage: r.read(filename)
sage: r.get('a')
'[1] 4'
```

```
>>> from sage.all import *
>>> filename = tmp_filename()
>>> f = open(filename, 'w')
>>> _ = f.write('a <- 2+2\n')
>>> f.close()
>>> r.read(filename)
>>> r.get('a')
'[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')
sage: 'grid' in r.eval('(.packages())')
True
sage: r.library('foobar')
Traceback (most recent call last):
...
ImportError: ...
```

```
>>> from sage.all import *
>>> r.library('grid')
>>> 'grid' in r.eval('(.packages())')
True
>>> r.library('foobar')
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` – string
- `value` – string

EXAMPLES:

```
sage: r.set('a', '2 + 3')
sage: r.get('a')
'[1] 5'
```

```
>>> from sage.all import *
>>> r.set('a', '2 + 3')
>>> r.get('a')
'[1] 5'
```

set_seed(*seed=None*)

Set the seed for R interpreter.

The seed should be an integer.

EXAMPLES:

```
sage: r = R()
sage: r.set_seed(1)
1
sage: r.sample("1:10", 5) # random
[1] 3 4 5 7 2
```

```
>>> from sage.all import *
>>> r = R()
>>> r.set_seed(Integer(1))
1
>>> r.sample("1:10", Integer(5)) # random
[1] 3 4 5 7 2
```

source(*s*)

Display the R source (if possible) about the function named s.

INPUT:

- `s` – string representing the function whose source code you want to see

OUTPUT: string; source code

EXAMPLES:

```
sage: print(r.source("c"))
function (...)  .Primitive("c")
```

```
>>> from sage.all import *
>>> print(r.source("c"))
function (...)  .Primitive("c")
```

version()

Return the version of R currently running.

OUTPUT: tuple of ints; string

EXAMPLES:

```
sage: r.version()                      # not tested
((3, 0, 1), 'R version 3.0.1 (2013-05-16)')
sage: rint, rstr = r.version()
sage: rint[0] >= 3
True
sage: rstr.startswith('R version')
True
```

```
>>> from sage.all import *
>>> r.version()                      # not tested
((3, 0, 1), 'R version 3.0.1 (2013-05-16)')
```

(continues on next page)

(continued from previous page)

```
>>> rint, rstr = r.version()
>>> rint[Integer(0)] >= Integer(3)
True
>>> rstr.startswith('R version')
True
```

class sage.interfaces.r.**RElement**(*parent*, *value*, *is_name=False*, *name=None*)

Bases: ExtraTabCompletion, *InterfaceElement*

dot_product(*other*)

Implement the notation self . other.

INPUT:

- self, other – R elements

OUTPUT: R element

EXAMPLES:

```
sage: c = r.c(1,2,3,4)
sage: c.dot_product(c.t())
     [,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])
sage: v.dot_product(v)
     [,1]
[1,]   74
```

```
>>> from sage.all import *
>>> c = r.c(Integer(1),Integer(2),Integer(3),Integer(4))
>>> c.dot_product(c.t())
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    2    4    6    8
[3,]    3    6    9   12
[4,]    4    8   12   16

>>> v = r([Integer(3),-Integer(1),Integer(8)])
>>> v.dot_product(v)
     [,1]
[1,]   74
```

is_string()

Tell whether this element is a string.

EXAMPLES:

```
sage: r('"abc"').is_string()
True
```

(continues on next page)

(continued from previous page)

```
sage: r([1,2,3]).is_string()
False
```

```
>>> from sage.all import *
>>> r('"abc"').is_string()
True
>>> r([Integer(1),Integer(2),Integer(3)]).is_string()
False
```

stat_model(*x*)

The tilde regression operator in R.

EXAMPLES:

```
sage: x = r([1,2,3,4,5])
sage: y = r([3,5,7,9,11])
sage: a = r.lm( y.tilde(x) ) # lm( y ~ x )
sage: d = a._sage_()
sage: d['DATA']['coefficients']['DATA'][1]
2
```

```
>>> from sage.all import *
>>> x = r([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)])
>>> y = r([Integer(3),Integer(5),Integer(7),Integer(9),Integer(11)])
>>> a = r.lm( y.tilde(x) ) # lm( y ~ x )
>>> d = a._sage_()
>>> d['DATA']['coefficients']['DATA'][Integer(1)]
2
```

tilde(*x*)

The tilde regression operator in R.

EXAMPLES:

```
sage: x = r([1,2,3,4,5])
sage: y = r([3,5,7,9,11])
sage: a = r.lm( y.tilde(x) ) # lm( y ~ x )
sage: d = a._sage_()
sage: d['DATA']['coefficients']['DATA'][1]
2
```

```
>>> from sage.all import *
>>> x = r([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)])
>>> y = r([Integer(3),Integer(5),Integer(7),Integer(9),Integer(11)])
>>> a = r.lm( y.tilde(x) ) # lm( y ~ x )
>>> d = a._sage_()
>>> d['DATA']['coefficients']['DATA'][Integer(1)]
2
```

class sage.interfaces.r.**RFunction**(*parent*, *name*, *r_name=None*)

Bases: *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
sage: type(length)
<class 'sage.interfaces.r.RFunction'>
sage: loads(dumps(length))
length
```

```
>>> from sage.all import *
>>> length = r.length
>>> type(length)
<class 'sage.interfaces.r.RFunction'>
>>> loads(dumps(length))
length
```

class `sage.interfaces.r.`**`RFunctionElement`**(*obj*, *name*)

Bases: *InterfaceFunctionElement*

`sage.interfaces.r.`**`is_RElement`**(*x*)

Return `True` if x is an element in an R interface.

INPUT:

- `x` – object

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.interfaces.r import is_RElement
sage: is_RElement(2)
doctest:...: DeprecationWarning: the function is_RElement is deprecated; use
↪isinstance(x, sage.interfaces.abc.RElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
sage: is_RElement(r(2))
True
```

```
>>> from sage.all import *
>>> from sage.interfaces.r import is_RElement
>>> is_RElement(Integer(2))
doctest:...: DeprecationWarning: the function is_RElement is deprecated; use
↪isinstance(x, sage.interfaces.abc.RElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
False
>>> is_RElement(r(Integer(2)))
True
```

`sage.interfaces.r.r_console()`

Spawn a new R command-line session.

EXAMPLES:

```
sage: r.console()                              # not tested
    R version 2.6.1 (2007-11-26)
    Copyright (C) 2007 The R Foundation for Statistical Computing
    ISBN 3-900051-07-0
    ...
```

```
>>> from sage.all import *
>>> r.console()                              # not tested
    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
((3, 0, 1), 'R version 3.0.1 (2013-05-16)')
sage: rint, rstr = r_version()
sage: rint[0] >= 3
True
sage: rstr.startswith('R version')
True
```

```
>>> from sage.all import *
>>> r_version()                               # not tested
((3, 0, 1), 'R version 3.0.1 (2013-05-16)')
>>> rint, rstr = r_version()
>>> rint[Integer(0)] >= Integer(3)
True
>>> rstr.startswith('R version')
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
sage: reduce_load_R()
R Interpreter
```

```
>>> from sage.all import *
>>> from sage.interfaces.r import reduce_load_R
>>> reduce_load_R()
R Interpreter
```

CHAPTER

FORTYONE

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@cwi.nl> (1993) – Initial interface by Robert Bradshaw (2007-08)

class sage.interfaces.rubik.**CubexSolver**

Bases: object

format_cube(*facets*)

solve(*facets*)

EXAMPLES:

```
sage: # optional - rubiks
sage: from sage.interfaces.rubik import *
sage: C = RubiksCube("R U")
sage: CubexSolver().solve(C.facets())
'R U'
sage: C = RubiksCube("R U F L B D")
sage: sol = CubexSolver().solve(C.facets()); sol
"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
True
sage: C = RubiksCube("R2 F'")
sage: CubexSolver().solve(C.facets())
"R' R' F'"
sage: C = RubiksCube().scramble()
sage: sol = CubexSolver().solve(C.facets())
sage: C == RubiksCube(sol)
True
```

```
>>> from sage.all import *
>>> # optional - rubiks
>>> from sage.interfaces.rubik import *
>>> C = RubiksCube("R U")
>>> CubexSolver().solve(C.facets())
'R U'
>>> C = RubiksCube("R U F L B D")
>>> sol = CubexSolver().solve(C.facets()); sol
"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'"
>>> RubiksCube(sol) == C
True
>>> C = RubiksCube("R2 F'")
>>> CubexSolver().solve(C.facets())
"R' R' F'"
>>> C = RubiksCube().scramble()
>>> sol = CubexSolver().solve(C.facets())
>>> C == RubiksCube(sol)
True
```

class sage.interfaces.rubik.**DikSolver**

Bases: object

facet_map = [1, 2, 3, 4, 0, 5, 6, 7, 8, 9, 10, 11, 17, 18, 19, 25, 26, 27, 33, 34, 35, 12, 0, 13, 20, 0, 21, 28, 0, 29, 36, 0, 37, 14, 15, 16, 22, 23, 24, 30, 31, 32, 38, 39, 40, 41, 42, 43, 44, 0, 45, 46, 47, 48]

format_cube(*facets*)

rot_map = {'B': 'U', 'D': 'B', 'F': 'D', 'L': 'L', 'R': 'R', 'U': 'F'}

solve(*facets*, *timeout=10*, *extra_time=2*)

EXAMPLES:

```
sage: # optional - rubiks
sage: from sage.interfaces.rubik import *
sage: C = RubiksCube().move("R U")
sage: DikSolver().solve(C.facets())
'R U'
sage: C = RubiksCube().move("R U F L B D")
sage: DikSolver().solve(C.facets())
'R U F L B D'
sage: C = RubiksCube().move("R2 F'")
sage: DikSolver().solve(C.facets())
"R2 F'"
```

```
>>> from sage.all import *
>>> # optional - rubiks
>>> from sage.interfaces.rubik import *
>>> C = RubiksCube().move("R U")
>>> DikSolver().solve(C.facets())
```

(continues on next page)

(continued from previous page)

```
'R U'
>>> C = RubiksCube().move("R U F L B D")
>>> DikSolver().solve(C.facets())
'R U F L B D'
>>> C = RubiksCube().move("R2 F'")
>>> DikSolver().solve(C.facets())
"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: # optional - rubiks
sage: from sage.interfaces.rubik import *
sage: solver = DikSolver()
sage: solver = OptimalSolver()      # long time (28s on sage.math, 2012)
Initializing tables...
Done.
sage: C = RubiksCube("R U")
sage: solver.solve(C.facets())
'R  U'
sage: C = RubiksCube("R U F L B D")
sage: solver.solve(C.facets())
'R  U  F  L  B  D'
sage: C = RubiksCube("R2 D2")
sage: solver.solve(C.facets())
'R2 D2'
```

```
>>> from sage.all import *
>>> # optional - rubiks
>>> from sage.interfaces.rubik import *
>>> solver = DikSolver()
>>> solver = OptimalSolver()      # long time (28s on sage.math, 2012)
Initializing tables...
Done.
>>> C = RubiksCube("R U")
>>> solver.solve(C.facets())
'R  U'
```

(continues on next page)

(continued from previous page)

```
>>> C = RubiksCube("R U F L B D")
>>> solver.solve(C.facets())
'R  U  F  L  B  D'
>>> C = RubiksCube("R2 D2")
>>> solver.solve(C.facets())
'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
```

```
>>> from sage.all import *
>>> from sage.interfaces.rubik import SingNot
>>> SingNot("acb") == SingNot("ACB")
True
>>> SingNot("acb") == SingNot("bca")
False
```

CHAPTER

FORTYTWO

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: ExtraTabCompletion, *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()
```

```
>>> from sage.all import *
>>> s = Sage()
```

Create the element 2 in our new copy of Sage, and cube it.

```
sage: a = s(2)
sage: a^3
8
```

```
>>> from sage.all import *
>>> a = s(Integer(2))
>>> a**Integer(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))
```

```
>>> from sage.all import *
>>> V = s('QQ^4')
>>> V.gens()
((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))
```

Note that V is a 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)
sage: g.parent()
Sage
```

```
>>> from sage.all import *
>>> type(V)
<class 'sage.interfaces.sage0.SageElement'>
>>> V.parent()
Sage
>>> g = V.gen(0);  g
(1, 0, 0, 0)
>>> 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
```

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

```
>>> from sage.all import *
>>> x = Integer(10)
>>> s('x = 5')
5
>>> x
```

(continues on next page)

(continued from previous page)

```
10
>>> 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
```

```
>>> from sage.all import *
>>> s3 = s('Sage()')
>>> a = s3(Integer(10))
>>> 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
```

```
>>> from sage.all import *
>>> _ = s.eval('%s.eval("x=8")' % s3.name())
>>> s3('"x"')
8
>>> s('x')
5
>>> 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 Issue #10539.

EXAMPLES:

```
sage: sage0.set('x', '2')
sage: sage0.get('x')
'2'
sage: sage0.clear('x')
sage: 'NameError' in sage0.get('x')
True
```

```
>>> from sage.all import *
>>> sage0.set('x', '2')
>>> sage0.get('x')
'2'
>>> sage0.clear('x')
>>> '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.                        |
----------------------------------------------------------------------
...
```

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

```
>>> from sage.all import *
>>> sage0.cputime()     # random output
1.3530439999999999
>>> sage0('factor(2^157-1)')
852133201 * 60726444167 * 1654058017289 * 2134387368610417
>>> sage0.cputime()     # random output
1.6462939999999999
```

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

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'
```

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

```
>>> from sage.all import *
>>> sage0.set('x', '2')
>>> sage0.get('x')
'2'
```

new(*x*)

EXAMPLES:

```
sage: sage0.new(2)
2
sage: _.parent()
Sage
```

```
>>> from sage.all import *
>>> sage0.new(Integer(2))
2
>>> _.parent()
Sage
```

preparse(*x*)

Return the preparsed version of the string s.

EXAMPLES:

```
sage: sage0.preparse('2+2')
'Integer(2)+Integer(2)'
```

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

```
>>> from sage.all import *
>>> sage0.set('x', '2')
>>> sage0.get('x')
'2'
```

version ()

EXAMPLES:

```
sage: sage0.version()
'SageMath version ..., Release Date: ...'
sage: sage0.version() == version()
True
```

```
>>> from sage.all import *
>>> sage0.version()
'SageMath version ..., Release Date: ...'
>>> sage0.version() == version()
True
```

class sage.interfaces.sage0.**SageElement** (*parent*, *value*, *is_name=False*, *name=None*)

Bases: *ExpectElement*

class sage.interfaces.sage0.**SageFunction** (*obj*, *name*)

Bases: *FunctionElement*

sage.interfaces.sage0.**reduce_load_Sage** ()

EXAMPLES:

```
sage: from sage.interfaces.sage0 import reduce_load_Sage
sage: reduce_load_Sage()
Sage
```

```
>>> from sage.all import *
>>> from sage.interfaces.sage0 import reduce_load_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
```

(continues on next page)

(continued from previous page)

```
sage: half.parent()
Sage
```

```
>>> from sage.all import *
>>> from sage.interfaces.sage0 import reduce_load_element
>>> s = dumps(Integer(1)/Integer(2))
>>> half = reduce_load_element(s); half
1/2
>>> 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.                        |
----------------------------------------------------------------------
...
```

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

```
>>> from sage.all import *
>>> from sage.interfaces.sage0 import sage0_version
>>> sage0_version() == version()
True
```

CHAPTER

FORTYTHREE

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: # optional - scilab
sage: scilab.eval('2+2')
'ans  =\n \n    4.'
sage: scilab('2+2')
4.
sage: a = scilab(10)
sage: a**10
1.000D+10
```

```
>>> from sage.all import *
>>> # optional - scilab
>>> scilab.eval('2+2')
'ans  =\n \n    4.'
>>> scilab('2+2')
4.
>>> a = scilab(Integer(10))
>>> a**Integer(10)
1.000D+10
```

Tutorial based the MATLAB interface tutorial:

EXAMPLES:

```
sage: # optional - scilab
sage: scilab('4+10')
14.
sage: scilab('date')
15-Feb-2010
sage: scilab('5*10 + 6')
56.
sage: scilab('(6+6)/3')
4.
```

(continues on next page)

(continued from previous page)

```
sage: scilab('9')^2
81.
sage: a = scilab(10); b = scilab(20); c = scilab(30)
sage: avg = (a+b+c)/3
sage: avg
20.
sage: parent(avg)
Scilab

sage: # optional - scilab
sage: my_scalar = scilab('3.1415')
sage: my_scalar
3.1415
sage: my_vector1 = scilab('[1,5,7]')
sage: my_vector1
1.    5.    7.
sage: my_vector2 = scilab('[1;5;7]')
sage: my_vector2
1.
5.
7.
sage: my_vector1 * my_vector2
75.

sage: # optional - scilab
sage: row_vector1 = scilab('[1 2 3]')
sage: row_vector2 = scilab('[3 2 1]')
sage: matrix_from_row_vec = scilab('[%s; %s]'%(row_vector1.name(), row_vector2.
↪name()))
sage: matrix_from_row_vec
1.    2.    3.
3.    2.    1.

sage: # optional - scilab
sage: column_vector1 = scilab('[1;3]')
sage: column_vector2 = scilab('[2;8]')
sage: matrix_from_col_vec = scilab('[%s %s]'%(column_vector1.name(), column_vector2.
↪name()))
sage: matrix_from_col_vec
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
```

(continues on next page)

(continued from previous page)

```
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: # optional - scilab
sage: my_vector1 = scilab('[1,5,7]')
sage: my_vector1(1)
1.
sage: my_vector1(2)
5.
sage: my_vector1(3)
7.
```

```
>>> from sage.all import *
>>> # optional - scilab
>>> scilab('4+10')
14.
>>> scilab('date')
15-Feb-2010
>>> scilab('5*10 + 6')
56.
>>> scilab('(6+6)/3')
4.
>>> scilab('9')**Integer(2)
81.
>>> a = scilab(Integer(10)); b = scilab(Integer(20)); c = scilab(Integer(30))
>>> avg = (a+b+c)/Integer(3)
>>> avg
20.
>>> parent(avg)
Scilab

>>> # optional - scilab
>>> my_scalar = scilab('3.1415')
>>> my_scalar
3.1415
>>> my_vector1 = scilab('[1,5,7]')
>>> my_vector1
1.    5.    7.
>>> my_vector2 = scilab('[1;5;7]')
>>> my_vector2
1.
5.
7.
>>> my_vector1 * my_vector2
75.
```

(continues on next page)

(continued from previous page)

```
>>> # optional - scilab
>>> row_vector1 = scilab('[1 2 3]')
>>> row_vector2 = scilab('[3 2 1]')
>>> matrix_from_row_vec = scilab('[%s; %s]'%(row_vector1.name(), row_vector2.name()))
>>> matrix_from_row_vec
1.    2.    3.
3.    2.    1.

>>> # optional - scilab
>>> column_vector1 = scilab('[1;3]')
>>> column_vector2 = scilab('[2;8]')
>>> matrix_from_col_vec = scilab('[%s %s]'%(column_vector1.name(), column_vector2.
→name()))
>>> matrix_from_col_vec
1.    2.
3.    8.

>>> my_matrix = scilab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')     # optional -␣
→scilab
>>> my_matrix                                          # optional - scilab
    8.     12.    19.
    7.     3.     2.
    12.    4.     23.
    8.     1.     1.

>>> combined_matrix = scilab('[%s, %s]'%(my_matrix.name(), my_matrix.name()))        ␣
→                                # optional - scilab
>>> 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.

>>> tm = scilab('0.5:2:10')                            # optional - scilab
>>> tm                                                 # optional - scilab
0.5    2.5    4.5    6.5    8.5

>>> # optional - scilab
>>> my_vector1 = scilab('[1,5,7]')
>>> my_vector1(Integer(1))
1.
>>> my_vector1(Integer(2))
5.
>>> my_vector1(Integer(3))
7.
```

Matrix indexing works as follows:

```
sage: my_matrix = scilab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')      # optional -␣
→scilab
sage: my_matrix(3,2)                                   # optional - scilab
```

(continues on next page)

(continued from previous page)

```
4.
```

```
>>> from sage.all import *
>>> my_matrix = scilab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')      # optional -␣
↪scilab
>>> my_matrix(Integer(3),Integer(2))                                  # optional -␣
↪scilab
4.
```

One can also use square brackets:

```
sage: my_matrix[3,2]                                  # optional - scilab
4.
```

```
>>> from sage.all import *
>>> my_matrix[Integer(3),Integer(2)]                                  # optional -␣
↪scilab
4.
```

Setting using parenthesis cannot work (because of how the Python language works). Use square brackets or the set function:

```
sage: # optional - scilab
sage: my_matrix = scilab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')
sage: my_matrix.set(2,3, 1999)
sage: my_matrix
       8.          12.          19.
       7.           3.        1999.
      12.           4.          23.
       8.           1.           1.
sage: my_matrix[2,3] = -126
sage: my_matrix
       8.          12.          19.
       7.           3.       - 126.
      12.           4.          23.
       8.           1.           1.
```

```
>>> from sage.all import *
>>> # optional - scilab
>>> my_matrix = scilab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')
>>> my_matrix.set(Integer(2),Integer(3), Integer(1999))
>>> my_matrix
       8.          12.          19.
       7.           3.        1999.
      12.           4.          23.
       8.           1.           1.
>>> my_matrix[Integer(2),Integer(3)] = -Integer(126)
>>> my_matrix
       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: *Expect*

Interface to the Scilab interpreter.

EXAMPLES:

```
sage: # optional - scilab
sage: a = scilab('[ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]')
sage: b = scilab('[ 1; 3; 13]')
sage: c = a * b
sage: print(c)
  30.
  122.
  505.
```

```
>>> from sage.all import *
>>> # optional - scilab
>>> a = scilab('[ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]')
>>> b = scilab('[ 1; 3; 13]')
>>> c = a * b
>>> print(c)
  30.
  122.
  505.
```

console()

Starts Scilab console.

EXAMPLES:

```
sage: scilab.console()          # optional - scilab; not tested
```

```
>>> from sage.all import *
>>> scilab.console()          # optional - scilab; not tested
```

eval(*command*, **args*, ***kwds*)

Evaluates commands.

EXAMPLES:

```
sage: scilab.eval("5")                      # optional - scilab
'ans  =
```

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

```
>>> from sage.all import *
>>> scilab.eval('b=124;')                      # optional - scilab
''
>>> 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: string that evaluates to a 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]'
```

```
>>> from sage.all import *
>>> M33 = MatrixSpace(QQ,Integer(3),Integer(3))              # optional -␣
↪scilab
>>> A = M33([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
↪Integer(6),Integer(7),Integer(8),Integer(0)])            # optional - scilab
>>> 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
'
```

```
>>> from sage.all import *
>>> scilab.set('a', Integer(123))         # optional - scilab
```

(continues on next page)

(continued from previous page)

```
>>> scilab.get('a')                # optional - scilab
'
```

123.'

set_seed(*seed=None*)

Set the seed for gp interpreter.

The seed should be an integer.

EXAMPLES:

```
sage: # optional - scilab
sage: from sage.interfaces.scilab import Scilab
sage: s = Scilab()
sage: s.set_seed(1)
1
sage: [s.rand() for i in range(5)]
[

     0.6040239,

     0.0079647,

     0.6643966,

     0.9832111,

     0.5321420]
```

```
>>> from sage.all import *
>>> # optional - scilab
>>> from sage.interfaces.scilab import Scilab
>>> s = Scilab()
>>> s.set_seed(Integer(1))
1
>>> [s.rand() for i in range(Integer(5))]
[
<BLANKLINE>
     0.6040239,
<BLANKLINE>
     0.0079647,
<BLANKLINE>
     0.6643966,
<BLANKLINE>
     0.9832111,
<BLANKLINE>
     0.5321420]
```

version()

Return the version of the Scilab software used.

EXAMPLES:

```
sage: scilab.version()                       # optional - scilab
'scilab-...'
```

```
>>> from sage.all import *
>>> scilab.version()                       # optional - scilab
'scilab-...'
```

whos (*name=None*, *typ=None*)

Return 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...'
```

```
>>> from sage.all import *
>>> scilab.whos("core")                     # optional - scilab
'Name                     Type           Size           Bytes...'
>>> scilab.whos(typ='function')             # optional - scilab
'Name                     Type           Size           Bytes...'
```

class sage.interfaces.scilab.**ScilabElement** (*parent*, *value*, *is_name=False*, *name=None*)

Bases: *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
'
```

```
>>> from sage.all import *
>>> scilab.set('c', Integer(125))          # optional - scilab
>>> scilab.get('c')               # optional - scilab
'
```

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
        ___________________________________________
```

(continues on next page)

(continued from previous page)

```
                       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
```

```
>>> from sage.all import *
>>> from sage.interfaces.scilab import scilab_console  # optional - scilab
>>> 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-...'
```

```
>>> from sage.all import *
>>> from sage.interfaces.scilab import scilab_version  # optional - scilab
>>> scilab_version()    # optional - scilab
'scilab-...'
```

CHAPTER

FORTYFOUR

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
```

```
>>> from sage.all import *
>>> R1 = singular.ring(Integer(0), '(x,y)', 'dp')
>>> R1
```

(continues on next page)

(continued from previous page)

```
polynomial ring, over a field, global ordering
// coefficients: QQ...
// number of vars : 2
//        block   1 : ordering dp
//                  : names    x y
//        block   2 : ordering C
>>> f = singular('9x16 - 18x13y2 - 9x12y3 + 9x10y4 - 18x11y2 + 36x8y4 + 18x7y5 -␣
↪18x5y6 + 9x6y4 - 18x3y6 - 9x2y7 + 9y8')
>>> 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
>>> 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
```

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

```
>>> from sage.all import *
>>> F[Integer(1)]
9,
x^6-2*x^3*y^2-x^2*y^3+y^4,
-x^5+y^2
>>> F[Integer(1)][Integer(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
```

(continues on next page)

(continued from previous page)

```
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
```

```
>>> from sage.all import *
>>> 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
>>> F[Integer(1)][Integer(2)].sage()
x^6 - 2*x^3*y^2 - x^2*y^3 + y^4
>>> 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
```

```
>>> from sage.all import *
>>> R2 = singular.ring(Integer(0), '(x,y,z)', 'lp')
>>> a = singular.new('3x2*(x+y)')
>>> b = singular.new('9x*(y2-x2)')
>>> g = a.gcd(b)
>>> 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
```

```
>>> from sage.all import *
>>> R3 = singular.ring(Integer(0), '(a,b,c,d)', 'lp')
>>> 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'])
>>> I2 = I.groebner()
>>> I2
c^2*d^6-c^2*d^2-d^4+1,
```

(continues on next page)

(continued from previous page)

```
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'])
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
```

```
>>> from sage.all import *
>>> R  = singular.ring(Integer(0), '(x0,x1,x2)', 'lp')
>>> 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'])
>>> I2 = I1.groebner()
>>> 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
>>> 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
```

```
>>> from sage.all import *
>>> R = singular.ring(Integer(0), '(x,y,z)', 'dp')
>>> f = singular('x3+y3+(x-y)*x2y2+z2')
>>> f
x^3*y^2-x^2*y^3+x^3+y^3+z^2
```

(continues on next page)

(continued from previous page)

```
>>> R1 = singular.ring(Integer(0), '(x,y,z)', 'ds')
>>> f = R.fetch(f)
>>> 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
```

```
>>> from sage.all import *
>>> _=singular.LIB('sing.lib')      # assign to _ to suppress printing
>>> 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
```

```
>>> from sage.all import *
>>> H = f.jacob().jacob()
>>> 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
>>> 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]
>>> 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
>>> 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,
```

(continues on next page)

(continued from previous page)

```
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,
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
```

```
>>> from sage.all import *
>>> H.minor(Integer(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,
0,
0,
0
>>> H.minor(Integer(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
```

```
>>> from sage.all import *
>>> _=singular.eval('option(redSB)')
>>> H.minor(Integer(1)).groebner()
1
```

44.3 Computing the Genus

We compute the projective genus of ideals that define curves over **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('y9 - x2*(x-1)^9 + x')
sage: i2.genus()
40
```

```
>>> from sage.all import *
>>> singular.lib('normal.lib')
>>> R = singular.ring(Integer(0),'(x,y)','dp')
>>> i2 = singular.ideal('y9 - x2*(x-1)^9 + x')
>>> i2.genus()
40
```

Note that the genus can be much smaller than the degree:

```
sage: i = singular.ideal('y9 - x2*(x-1)^9')
sage: i.genus()
0
```

```
>>> from sage.all import *
>>> i = singular.ideal('y9 - x2*(x-1)^9')
>>> 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
//                          : 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:
...
```

```
>>> from sage.all import *
>>> singular.lib('polylib.lib')
>>> singular.ring(Integer(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
//                          : names    a b c d e f
//        block   2 : ordering C
>>> I = singular.ideal('cyclic(6)')
>>> 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,
...
```

```
>>> from sage.all import *
>>> singular.quit()
>>> singular.lib('polylib.lib'); R = singular.ring(Integer(32003), '(a,b,c,d,e,f)',
↪'lp')
>>> I = singular.ideal('cyclic(6)')
>>> 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)
```

```
>>> from sage.all import *
>>> t = '"%s"'%Integer(10)**Integer(15000)   # 15 thousand character string (note␣
↪that normal Singular input must be at most 10000)
```

(continues on next page)

(continued from previous page)

```
>>> a = singular.eval(t)
>>> 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: ExtraTabCompletion, *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
```

```
>>> from sage.all import *
>>> R = singular.ring(Integer(0), '(x0,x1,x2)', 'lp')
>>> 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'])
>>> 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)
```

```
>>> from sage.all import *
>>> singular.lib('sing.lib')
>>> 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')
```

```
>>> from sage.all import *
>>> singular.set('int', 'x', '2')
>>> singular.get('x')
'2'
>>> 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`'
```

```
>>> from sage.all import *
>>> singular.get('x')
'2'
>>> a = singular(Integer(3))
>>> 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<
```

(continues on next page)

(continued from previous page)

```
     by: G.-M. Greuel, G. Pfister, H. Schoenemann        \   Nov 2007
FB Mathematik der Universitaet, D-67653 Kaiserslautern    \
```

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

Return 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
```

```
>>> from sage.all import *
>>> t = singular.cputime()
>>> R = singular.ring(Integer(0), '(x0,x1,x2)', 'lp')
>>> 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'])
>>> gb = I.groebner()
>>> singular.cputime(t) #random
0.02
```

current_ring()

Return 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 ip
//                  : 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 ip
```

(continues on next page)

(continued from previous page)

```
//                  : names    x y z
//        block   2 : ordering C
```

```
>>> from sage.all import *
>>> r = PolynomialRing(GF(Integer(127)),Integer(3),'xyz', order='invlex')
>>> r._singular_()
polynomial ring, over a field, global ordering
// coefficients: ZZ/127...
// number of vars : 3
//        block   1 : ordering ip
//                  : names    x y z
//        block   2 : ordering C
>>> singular.current_ring()
polynomial ring, over a field, global ordering
// coefficients: ZZ/127...
// number of vars : 3
//        block   1 : ordering ip
//                  : names    x y z
//        block   2 : ordering C
```

current_ring_name()

Return 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
```

```
>>> from sage.all import *
>>> r = PolynomialRing(GF(Integer(127)),Integer(3),'xyz')
>>> r._singular_().name() == singular.current_ring_name()
True
```

eval(*x*, *allow_semicolon=True*, *strip=True*, ***kwds*)

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'
```

```
>>> from sage.all import *
>>> singular.eval('2 > 1')
'1'
>>> 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()))
'...// dimension (affine) = 0\n//
degree (affine) = 8'
```

```
>>> from sage.all import *
>>> r = singular.ring(Integer(0),'(x,y,z)','dp')
>>> i = singular.ideal(['x^2','y^2','z^2'])
>>> s = i.std()
>>> singular.eval('hilb(%s)'%(s.name()))
'...// dimension (affine) = 0\n//
degree (affine) = 8'
```

```
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(1)
sage: o = singular.eval('hilb(%s)'%(s.name()))
...// dimension (affine) = 0
// degree (affine)  = 8
```

```
>>> from sage.all import *
>>> from sage.misc.verbose import set_verbose
>>> set_verbose(Integer(1))
>>> o = singular.eval('hilb(%s)'%(s.name()))
...// 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()
...// dimension (affine) = 0
// degree (affine)  = 8
// ** right side is not a datum, assignment ignored
...
```

```
>>> from sage.all import *
>>> o = s.hilb()
...// 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()
```

```
>>> from sage.all import *
>>> set_verbose(Integer(0))
>>> o = s.hilb()
```

get(*var*)

Get string representation of variable named var.

EXAMPLES:

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

```
>>> from sage.all import *
>>> singular.set('int', 'x', '2')
>>> 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
```

```
>>> from sage.all import *
>>> _ = singular.eval("ring R=0,(x0,x1,x2),lp")
>>> 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'])
>>> 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,
```

(continues on next page)

(continued from previous page)

```
x0*x1-x0*x2-x1*x2,
x0^2*x2-x0*x2^2-x1*x2^2
```

```
>>> from sage.all import *
>>> i2 = singular.ideal('groebner(%s);'%i1.name())
>>> 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)
```

```
>>> from sage.all import *
>>> singular.lib('sing.lib')
>>> singular.lib('sing.lib', reload=True)
```

list(*x*)

Create a list in Singular from a Sage list `x`.

EXAMPLES:

```
sage: singular.list([1,2])
[1]:
   1
[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]:
```

(continues on next page)

(continued from previous page)

```
   2
[3]:
   [1]:
      x
   [2]:
      _[1]=x
      _[2]=y
```

```
>>> from sage.all import *
>>> singular.list([Integer(1),Integer(2)])
[1]:
   1
[2]:
   2

>>> singular.list([Integer(1),Integer(2),[Integer(3),Integer(4)]])
[1]:
   1
[2]:
   2
[3]:
   [1]:
      3
   [2]:
      4

>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> singular.list([Integer(1),Integer(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
```

```
>>> from sage.all import *
>>> singular.list([Integer(1),Integer(2),'"hi"'])
[1]:
```

(continues on next page)

(continued from previous page)

```
    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)
```

```
>>> from sage.all import *
>>> singular.lib('sing.lib')
>>> 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')
sage: A
1,2,
3,4,
5,6
sage: A.gauss_col()
2,-1,
1,0,
0,1
```

```
>>> from sage.all import *
>>> singular.lib("matrix")
>>> R = singular.ring(Integer(0), '(x,y,z)', 'dp')
>>> A = singular.matrix(Integer(3),Integer(2),'1,2,3,4,5,6')
>>> A
1,2,
3,4,
5,6
>>> 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
```

```
>>> from sage.all import *
>>> singular.option()
//options: redefine loadLib usage prompt
>>> singular.option('get')
0,
10321
>>> old_options = _
>>> singular.option('noredefine')
>>> singular.option()
//options: loadLib usage prompt
>>> singular.option('set', old_options)
>>> 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` – 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 $\mathbf{Q}[x, y, z]$ 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
```

```
>>> from sage.all import *
>>> R = singular.ring(Integer(0), '(x,y,z)', 'dp')
>>> 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')
```

```
>>> from sage.all import *
>>> R1 = singular.ring(Integer(32003), '(x,y,z)', 'dp')
>>> R2 = singular.ring(Integer(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')
```

```
>>> from sage.all import *
>>> R3 = singular.ring(Integer(7), '(x(1..10))', 'ds')
```

This is a polynomial ring over the transcendental extension $\mathbf{Q}(a)$ of $\mathbf{Q}$:

```
sage: R4 = singular.ring('(0,a)', '(mu,nu)', 'lp')
```

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

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

```
>>> from sage.all import *
>>> R6 = singular.ring('(real,50)', '(a,b)', 'lp')
>>> 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
```

```
>>> from sage.all import *
>>> R = singular.ring(Integer(7), '(a,b)', 'ds')
>>> S = singular.ring('real', '(a,b)', 'lp')
>>> singular.new('10*a')
(1.000e+01)*a
>>> R.set_ring()
>>> 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'
```

```
>>> from sage.all import *
>>> singular.set('int', 'x', '2')
>>> 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'
```

```
>>> from sage.all import *
>>> a = singular(Integer(3))
>>> n = a.name()
>>> del a
>>> singular.eval(n)
'3'
>>> singular.set('int', 'y', '5')
>>> singular.eval('defined(%s)'%n)
'0'
```

set_ring(R)

Set 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
```

```
>>> from sage.all import *
>>> R = singular.ring(Integer(7), '(a,b)', 'ds')
>>> S = singular.ring('real', '(a,b)', 'lp')
>>> 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
>>> singular.set_ring(R)
```

(continues on next page)

(continued from previous page)

```
>>> 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]
```

```
>>> from sage.all import *
>>> s = Singular()
>>> s.set_seed(Integer(1))
1
>>> [s.random(Integer(1),Integer(10)) for i in range(Integer(5))]
[8, 10, 4, 9, 1]
```

setring(*R*)

Set 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
```

```
>>> from sage.all import *
>>> R = singular.ring(Integer(7), '(a,b)', 'ds')
>>> S = singular.ring('real', '(a,b)', 'lp')
>>> 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
>>> singular.set_ring(R)
>>> 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*)

Create a Singular string from a Sage string. Note that the Sage string has to be "double-quoted".

EXAMPLES:

```
sage: singular.string('"Sage"')
Sage
```

```
>>> from sage.all import *
>>> singular.string('"Sage"')
Sage
```

version()

Return the version of Singular being used.

EXAMPLES:

```
sage: singular.version()
"Singular ... version 4...
```

```
>>> from sage.all import *
>>> singular.version()
"Singular ... version 4...
```

class sage.interfaces.singular.**SingularElement**(*parent*, *type*, *value*, *is_name=False*)

Bases: `ExtraTabCompletion`, *ExpectElement*, *SingularElement*

EXAMPLES:

```
sage: a = singular(2)
sage: loads(dumps(a))
2
```

```
>>> from sage.all import *
>>> a = singular(Integer(2))
>>> 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'
```

```
>>> from sage.all import *
>>> P = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = P._first_
↪ngens(3)
>>> I = Ideal([z**Integer(2), y*z, y**Integer(2), x*z, x*y, x**Integer(2)])
>>> Ibar = I._singular_()
>>> Ibar.attrib('isSB')
0
>>> singular.eval('vdim(%s)'%Ibar.name()) # sage7 name is random
// ** sage7 is no standard basis
4
>>> Ibar.attrib('isSB',Integer(1))
>>> 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
```

```
>>> from sage.all import *
>>> singular('"abc"').is_string()
True
>>> 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']
```

```
>>> from sage.all import *
>>> R=singular.ring(Integer(0),'(x,y)','dp')
>>> RL = R.ringlist()
>>> 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)
```

```
>>> from sage.all import *
>>> singular.eval('ring r1 = (9,x),(a,b,c,d,e,f),(M((1,2,3,0)),wp(2,3),lp)')
''
>>> R = singular('r1').sage_global_ring()
>>> R
Multivariate Polynomial Ring in a, b, c, d, e, f over Finite Field in x of␣
↪size 3^2
>>> R.term_order()
```

(continues on next page)

(continued from previous page)

```
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
```

```
>>> from sage.all import *
>>> singular.eval('ring r2 = (0,x),(a,b,c),dp')
''
>>> 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
```

```
>>> from sage.all import *
>>> singular.eval('ring r3 = (3,z),(a,b,c),dp')
''
>>> singular.eval('minpoly = 1+z+z2+z3+z4')
''
>>> 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
```

```
>>> from sage.all import *
>>> singular.eval('ring r4 = (real,20),(a,b,c),dp')
''
>>> 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')
1.00000000000000*I
```

```
>>> from sage.all import *
>>> singular.eval('ring r5 = (complex,15,j),(a,b,c),dp')
''
>>> R = singular('r5').sage_global_ring(); R
Multivariate Polynomial Ring in a, b, c over Complex Field with 54 bits of␣
↪precision
>>> R.base_ring()('k')
Traceback (most recent call last):
...
ValueError: given string 'k' is not a complex number
>>> R.base_ring()('I')
1.00000000000000*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
```

```
>>> from sage.all import *
>>> singular.eval('ring r7 = (0,a), (x,y), dp')
''
>>> singular.eval('minpoly = a2 + 1')
''
>>> 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')
sage: Q.sage_global_ring()
```

(continues on next page)

(continued from previous page)

```
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)
```

```
>>> from sage.all import *
>>> singular.eval('ring r6 = (9,a), (x,y,z),lp')
''
>>> Q = singular('std(ideal(x^2,x+y^2+z^3))', type='qring')
>>> 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*)

Return 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` – boolean (default: `True`); 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]
```

```
>>> from sage.all import *
>>> R = singular.ring(Integer(0), '(x,y,z)', 'dp')
>>> A = singular.matrix(Integer(2),Integer(2))
>>> A.sage_matrix(ZZ)
[0 0]
[0 0]
>>> A.sage_matrix(RDF)
[0.0 0.0]
[0.0 0.0]
```

sage_poly(*R=None*, *kcache=None*)

Return 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
```

```
>>> from sage.all import *
>>> R = PolynomialRing(GF(Integer(2)**Integer(8),'a'), 'x,y')
>>> f = R('a^20*x^2*y+a^10+x')
>>> f._singular_().sage_poly(R) == f
True
>>> R = PolynomialRing(GF(Integer(2)**Integer(8),'a'), 'x', implementation=
↪'singular')
>>> f = R('a^20*x^3+x^2+a^10')
>>> 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
sage: singular(f)
x*y^3-1/9*x+1
sage: P(singular(f))
x*y^3 - 1/9*x + 1
```

```
>>> from sage.all import *
>>> P = PolynomialRing(QQ, Integer(2), names=('x', 'y',)); (x, y,) = P._first_
↪ngens(2)
>>> f = x*y**Integer(3) - Integer(1)/Integer(9) * x + Integer(1); f
x*y^3 - 1/9*x + 1
>>> singular(f)
x*y^3-1/9*x+1
>>> 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'
```

```
>>> from sage.all import *
>>> R = singular.ring(Integer(0),'(x,y)')
>>> f = singular('x^3 + 3*y^11 + 5')
>>> f
x^3+3*y^11+5
>>> f.sage_polystring()
'x**3+3*y**11+5'
```

sage_structured_str_list()

If `self` is a Singular list of lists of Singular elements, return 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
   [2]:
      [1]:
         C
      [2]:
         0
[4]:
   _[1]=0
sage: RL.sage_structured_str_list()
['0', ['x', 'y'], [['dp', '1,\n1'], ['C', '0']], '0']
```

```
>>> from sage.all import *
>>> R=singular.ring(Integer(0),'(x,y)','dp')
>>> RL=R.ringlist()
>>> RL
[1]:
   0
[2]:
   [1]:
      x
   [2]:
      y
[3]:
   [1]:
      [1]:
         dp
      [2]:
         1,1
   [2]:
      [1]:
         C
      [2]:
         0
[4]:
   _[1]=0
>>> RL.sage_structured_str_list()
['0', ['x', 'y'], [['dp', '1,\n1'], ['C', '0']], '0']
```

set_ring()

Set 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
```

```
>>> from sage.all import *
>>> R = singular.ring(Integer(7), '(a,b)', 'ds')
>>> S = singular.ring('real', '(a,b)', 'lp')
```

(continues on next page)

(continued from previous page)

```
>>> 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
>>> R.set_ring()
>>> 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()

Return 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'
```

```
>>> from sage.all import *
>>> R = PolynomialRing(GF(Integer(2)**Integer(8),'a'),Integer(2),'x')
>>> R._singular_().type()
'ring'
>>> fs = singular('x0^2','poly')
>>> 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: *ExpectFunction*

class sage.interfaces.singular.**SingularFunctionElement**(*obj*, *name*)

Bases: *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.

cri_hilb = re.compile('h')

crt_lne1 = re.compile('product criterion:(\\d+) chain criterion:(\\d+)')

crt_lne2 = re.compile('NF:(\\d+) product criterion:(\\d+), ext_product criterion:(\\d+)')

deg_lead = re.compile('\\d+')

flush()

EXAMPLES:

```
sage: from sage.interfaces.singular import SingularGBLogPrettyPrinter
sage: s3 = SingularGBLogPrettyPrinter(verbosity=3)
sage: s3.flush()
```

```
>>> from sage.all import *
>>> from sage.interfaces.singular import SingularGBLogPrettyPrinter
>>> s3 = SingularGBLogPrettyPrinter(verbosity=Integer(3))
>>> s3.flush()
```

global_pattern = re.compile('(\\[\\d+:\\d+\\]|s|-|\\.|h|H\\(\\d+\\)|\\(\\d+\\)|\\(S:\\d+\\)|\\d+|M\\[\\d+,[b,e]*\\d+\\]|b|e).*')

hig_corn = re.compile('H\\(\\d+\\)')

new_elem = re.compile('s')

non_mini = re.compile('e')

num_crit = re.compile('\\(\\d+\\)')

pat_sync = re.compile('1\\+(\\d+);')

red_betr = re.compile('b')

red_num = re.compile('\\(S:\\d+\\)')

red_para = re.compile('M\\[(\\d+),(\\d+)\\]')

red_post = re.compile('\\.')

red_zero = re.compile('-')

rng_chng = re.compile('\\[\\d+:\\d+\\]')

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 nonzero output elements.
```

```
>>> from sage.all import *
>>> from sage.interfaces.singular import SingularGBLogPrettyPrinter
>>> s3 = SingularGBLogPrettyPrinter(verbosity=Integer(3))
>>> s3.write("(S:1337)")
Performing complete reduction of 1337 elements.
>>> s3.write("M[389,12]")
Parallel reduction of 389 elements with 12 nonzero output elements.
```

sage.interfaces.singular.**get_docstring**(*name*, *prefix=False*, *code=False*)

Return the docstring for the function `name`.

INPUT:

- `name` – a Singular function name
- `prefix` – boolean (default: `False`); whether or not to include the prefix stating that what follows is from the Singular documentation.
- `code` – boolean (default: `False`); whether or not to format the result as a reStructuredText code block. This is intended to support the feature requested in Issue #11268.

OUTPUT:

A string describing the Singular function `name`. A `KeyError` is raised if the function was not found in the Singular documentation. If the "info" is not on the user's `PATH`, an `OSError` will be raised. If "info" was found but failed to execute, a `subprocess.CalledProcessError` will be raised instead.

EXAMPLES:

```
sage: from sage.interfaces.singular import get_docstring
sage: 'groebner' in get_docstring('groebner')   # needs_info
True
sage: 'standard.lib' in get_docstring('groebner')   # needs info
True
```

```
>>> from sage.all import *
>>> from sage.interfaces.singular import get_docstring
>>> 'groebner' in get_docstring('groebner')   # needs_info
True
>>> 'standard.lib' in get_docstring('groebner')   # needs info
True
```

The `prefix=True` form is used in Sage's generated docstrings:

```
sage: from sage.interfaces.singular import get_docstring
sage: print(get_docstring("factorize", prefix=True))   # needs info
The Singular documentation for "factorize" is given below.
...
```

```
>>> from sage.all import *
>>> from sage.interfaces.singular import get_docstring
>>> print(get_docstring("factorize", prefix=True))  # needs info
The Singular documentation for "factorize" is given below.
...
```

sage.interfaces.singular.**is_SingularElement**(*x*)

Return `True` if `x` is of type *SingularElement*.

This function is deprecated; use `isinstance()` (of *sage.interfaces.abc.SingularElement*) instead.

EXAMPLES:

```
sage: from sage.interfaces.singular import is_SingularElement
sage: is_SingularElement(singular(2))
doctest:...: DeprecationWarning: the function is_SingularElement is deprecated;␣
↪use isinstance(x, sage.interfaces.abc.SingularElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
True
sage: is_SingularElement(2)
False
```

```
>>> from sage.all import *
>>> from sage.interfaces.singular import is_SingularElement
>>> is_SingularElement(singular(Integer(2)))
doctest:...: DeprecationWarning: the function is_SingularElement is deprecated;␣
↪use isinstance(x, sage.interfaces.abc.SingularElement) instead
See https://github.com/sagemath/sage/issues/34804 for details.
True
>>> is_SingularElement(Integer(2))
False
```

sage.interfaces.singular.**reduce_load_Singular**()

EXAMPLES:

```
sage: from sage.interfaces.singular import reduce_load_Singular
sage: reduce_load_Singular()
Singular
```

```
>>> from sage.all import *
>>> from sage.interfaces.singular import reduce_load_Singular
>>> 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<
```

(continues on next page)

(continued from previous page)

```
     by: G.-M. Greuel, G. Pfister, H. Schoenemann        \   Nov 2007
FB Mathematik der Universitaet, D-67653 Kaiserslautern    \
```

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

```
>>> from sage.all import *
>>> singular.version()
"Singular ... version 4...
```

CHAPTER

FORTYFIVE

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_()
```

```
>>> from sage.all import *
>>> from sympy import Symbol, Si, Ci, Shi, Chi, sign
>>> sx = Symbol('x')
>>> assert sin_integral(x)._sympy_() == Si(sx)
>>> assert sin_integral(x) == Si(sx)._sage_()
>>> assert sinh_integral(x)._sympy_() == Shi(sx)
>>> assert sinh_integral(x) == Shi(sx)._sage_()
>>> assert cos_integral(x)._sympy_() == Ci(sx)
>>> assert cos_integral(x) == Ci(sx)._sage_()
>>> assert cosh_integral(x)._sympy_() == Chi(sx)
>>> assert cosh_integral(x) == Chi(sx)._sage_()
>>> assert sgn(x)._sympy_() == sign(sx)
>>> 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_()
```

```
>>> from sage.all import *
>>> from sympy import Function
>>> f = function('f')
>>> F = Function('f')
>>> assert f._sympy_() == F
>>> assert f == F._sage_()
```

sage.interfaces.sympy.**check_expression**(*expr*, *var_symbols*, *only_from_sympy=False*)

Do `eval(expr)` both in Sage and SymPy and other checks.

EXAMPLES:

```
sage: from sage.interfaces.sympy import check_expression
sage: check_expression("1.123*x", "x")
```

```
>>> from sage.all import *
>>> from sage.interfaces.sympy import check_expression
>>> 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_()
```

```
>>> from sage.all import *
>>> from sage.interfaces.sympy import sympy_init
>>> from sympy import Symbol, Abs
>>> sympy_init()
>>> 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()
```

```
>>> from sage.all import *
>>> from sage.interfaces.sympy import test_all
>>> test_all()
```

CHAPTER

FORTYSIX

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 show 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
    TEXFUNC 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
    TEXFUNC 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
    TEXFUNC  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
```

(continues on next page)

(continued from previous page)

```
  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
```

46.8 Texture and Color

46.8.1 Simple Texture Characteristics

The surface textures applied to an object drastically alter its overall appearance, making textures and color one of the most important topics in this manual. As with many other renderers, textures can be declared and associated with a name so that they may be used over and over again in a scene definition with less typing. If a texture is only need once, or it is unique to a particular object in the scene, then it may be declared along with the object it is applied to, and does not need a name.

The simplest texture definition is a solid color with no image mapping or procedural texture mapping. A solid color texture is defined by the `AMBIENT`, `DIFFUSE`, `SPECULAR`, `OPACITY` and `COLOR` parameters. The `AMBIENT` parameter defines the ambient lighting coefficient to be used when shading the object. Similarly, the `DIFFUSE` parameter is the relative contribution of the diffuse shading to the surface appearance. The `SPECULAR` parameter is the contribution from perfectly reflected rays, as if on a mirrored surface. `OPACITY` defines how transparent a surface is. An `OPACITY` value of 0.0 renders the object completely invisible. An `OPACITY` value of 1.0 makes the object completely solid, and non-transmissive. In general, the values for the ambient, diffuse, and specular parameters should add up to 1.0, if they don't then pixels may be over or underexposed quite easily. These parameters function in a manner similar to that of other ray tracers. The `COLOR` parameter is an RGB triple with each value ranging from 0.0 to 1.0 inclusive. If the RGB values stray from 0.0 to 1.0, results are undefined. In the case of solid textures, a final parameter, `TEXFUNC` is set to zero (integer).

46.8.2 Texture Declaration and Aliasing

To define a simple texture for use on several objects in a scene, the `TEXDEF` keyword is used. The `TEXDEF` keyword is followed by a case sensitive texture name, which will subsequently be used while defining objects. If many objects in a scene use the same texture through texture definition, a significant amount of memory may be saved since only one copy of the texture is present in memory, and its shared by all of the objects. Here is an example of a solid texture definition:

```
TEXDEF MyNewRedTexture
   AMBIENT 0.1 DIFFUSE 0.9 SPECULAR 0.0 OPACITY 1.0
   COLOR 1.0 0.0 0.0  TEXFUNC 0
```

When this texture is used in an object definition, it is referenced only by name. Be careful not to use one of the other keywords as a defined texture, this will probably cause the parser to explode, as I don't check for use of keywords as texture names.

When a texture is declared within an object definition, it appears in an identical format to the `TEXDEF` declaration, but the `TEXTURE` keyword is used instead of `TEXDEF`. If it is useful to have several names for the same texture (when you are too lazy to actually finish defining different variations of a wood texture for example, and just want to be approximately correct for example) aliases can be constructed using the `TEXALIAS` keyword, along with the alias name, and the original name. An example of a texture alias is:

```
TEXALIAS MyNewestRedTexture  MyNewRedTexture
```

This line would alias MyNewestRedTexture to be the same thing as the previously declared MyNewRedTexture. Note that the source texture must be declared before any aliases that use it.

46.8.3 Image Maps and Procedural Textures

Image maps and procedural textures very useful in making realistic looking scenes. A good image map can do as much for the realism of a wooden table as any amount of sophisticated geometry or lighting. Image maps are made by wrapping an image on to an object in one of three ways, a spherical map, a cylindrical map, and a planar map. Procedural textures are used in a way similar to the image maps, but they are on the fly and do not use much memory compared to the image maps. The main disadvantage of the procedural maps is that they must be hard-coded into RAY when it is compiled.

The syntax used for all texture maps is fairly simple to learn. The biggest problem with the way that the parser is written now is that the different mappings are selected by an integer, which is not very user friendly. I expect to rewrite this section of the parser sometime in the near future to alleviate this problem. When I rewrite the parser, I may also end up altering the parameters that are used to describe a texture map, and some of them may become optional rather than required.

Texture Mapping Functions

Value for TEXFUNC	Mapping and Texture Description
0	No special texture, plain shading
1	3D checkerboard function, like a Rubik's cube
2	Grit Texture, randomized surface color
3	3D marble texture, uses object's base color
4	3D wood texture, light and dark brown, not very good yet
5	3D gradient noise function (can't remember what it look like
6	Don't remember
7	Cylindrical Image Map, requires ppm filename
8	Spherical Image Map, requires ppm filename
9	Planar Image Map, requires ppm filename

Here's an example of a sphere, with a spherical image map applied to its surface:

```
SPHERE
  CENTER 2.0  0.0 5.0
  RAD 2.0
  TEXTURE
    AMBIENT  0.4 DIFFUSE  0.8 SPECULAR 0.0  OPACITY 1.0
    COLOR 1.0 1.0 1.0
    TEXFUNC 7 /cfs/johns/imaps/fire644.ppm
      CENTER 2.0 0.0 5.0
      ROTATE 0.0 0.0 0.0
      SCALE  2.0 -2.0 1.0
```

Basically, the image maps require the center, rotate and scale parameters so that you can position the image map on the object properly.

class sage.interfaces.tachyon.**TachyonRT**

Bases: SageObject

The Tachyon Ray Tracer.

Usage: `tachyon_rt(model, outfile='sage.png', verbose=1, block=True, extra_opts='')`

INPUT:

- `model` – string that describes a 3d model in the Tachyon modeling format. Type `sage.interfaces.tachyon?` for a description of this format.
- `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` – boolean (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')
```

(continues on next page)

(continued from previous page)

```
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...
```

```
>>> from sage.all import *
>>> from sage.interfaces.tachyon import TachyonRT
>>> tgen = Tachyon()
>>> tgen.texture('t1')
>>> tgen.sphere((Integer(0),Integer(0),Integer(0)),Integer(1),'t1')
>>> tgen.str()[Integer(30):Integer(40)]
'resolution'
>>> t = TachyonRT()
>>> import os
>>> 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*)

Return 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`.

version()

Return the version of the Tachyon raytracer being used.

CHAPTER

FORTYSEVEN

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):
#
#                  https://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` – 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` – 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.

EXAMPLES:

```
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))])
```

```
>>> from sage.all import *
>>> from sage.interfaces.tides import subexpressions_list, remove_constants
>>> __tmp__=var("a"); f = symbolic_expression([Integer(1)+cos(Integer(7))*a]).
↪function(a)
>>> l1, l2 = subexpressions_list(f)
>>> 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))])
>>> remove_constants(l1,l2)
>>> 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)],
[('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)])
```

```
>>> from sage.all import *
>>> from sage.interfaces.tides import (subexpressions_list, remove_repeated)
>>> __tmp__=var("a"); f = symbolic_expression([Integer(1) + a**Integer(2),␣
↪arcsin(a)]).function(a)
>>> l1, l2 = subexpressions_list(f)
>>> l1, l2
([a^2, a^2 + 1, a^2, -a^2, -a^2 + 1, sqrt(-a^2 + 1), arcsin(a)],
[('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)])
```

(continues on next page)

(continued from previous page)

```
>>> remove_repeated(l1, l2)
>>> 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` – 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))])
```

```
>>> from sage.all import *
>>> from sage.interfaces.tides import subexpressions_list
>>> var('x,y')
(x, y)
>>> __tmp__=var("x,y"); f = symbolic_expression([x**Integer(2)+y, cos(x)/log(y)]).
↪function(x,y)
>>> 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),
```

(continues on next page)

(continued from previous page)

```
('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)
([sin(a), cos(a), a^2, a^2 + 1, arctan(a)],
[('sin', a), ('cos', a), ('mul', a, a), ('add', 1, a^2), ('atan', a)])
```

```
>>> from sage.all import *
>>> __tmp__=var("a"); f = symbolic_expression([cos(a), arctan(a)]).function(a)
>>> from sage.interfaces.tides import subexpressions_list
>>> subexpressions_list(f)
([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)])
```

```
>>> from sage.all import *
>>> from sage.interfaces.tides import subexpressions_list
```

(continues on next page)

(continued from previous page)

```
>>> var('s,b,r')
(s, b, r)
>>> __tmp__=var("t,x,y,z"); f = symbolic_expression([s*(y-x),x*(r-z)-y,x*y-b*z]).
↪function(t,x,y,z)
>>> 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))])
```

```
>>> from sage.all import *
>>> var('x, y')
(x, y)
>>> __tmp__=var("x,y"); f = symbolic_expression([exp(x**Integer(2)+sin(y))]).
↪function(x,y)
>>> from sage.interfaces.tides import *
>>> subexpressions_list(f)
```

(continues on next page)

(continued from previous page)

```
([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))])
```

CHAPTER

FORTYEIGHT

INTERFACE TO THE SAGE CLEANER

Triva 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.

CHAPTER FORTYNINE

QUITTING INTERFACES

`sage.interfaces.quit.expect_quitall(verbose=False)`

EXAMPLES:

```
sage: sage.interfaces.quit.expect_quitall()
sage: gp.eval('a=10')                                                       #␣
↪needs sage.libs.pari
'10'
sage: gp('a')                                                               #␣
↪needs sage.libs.pari
10
sage: sage.interfaces.quit.expect_quitall()
sage: gp('a')                                                               #␣
↪needs sage.libs.pari
a
sage: sage.interfaces.quit.expect_quitall(verbose=True)                     #␣
↪needs sage.libs.pari
Exiting PARI/GP interpreter with PID ... running .../gp --fast --emacs --quiet --
↪stacksize 10000000
```

```
>>> from sage.all import *
>>> sage.interfaces.quit.expect_quitall()
>>> gp.eval('a=10')                                                       #␣
↪needs sage.libs.pari
'10'
>>> gp('a')                                                               #␣
↪needs sage.libs.pari
10
>>> sage.interfaces.quit.expect_quitall()
>>> gp('a')                                                               #␣
↪needs sage.libs.pari
a
>>> sage.interfaces.quit.expect_quitall(verbose=True)                     #␣
↪needs sage.libs.pari
Exiting PARI/GP interpreter with PID ... running .../gp --fast --emacs --quiet --
↪stacksize 10000000
```

`sage.interfaces.quit.invalidate_all()`

Invalidate all of the expect interfaces.

This is used, e.g., by the fork-based `@parallel` decorator.

EXAMPLES:

```
sage: # needs sage.libs.pari sage.symbolic
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.)
```

```
>>> from sage.all import *
>>> # needs sage.libs.pari sage.symbolic
>>> a = maxima(Integer(2)); b = gp(Integer(3))
>>> a, b
(2, 3)
>>> sage.interfaces.quit.invalidate_all()
>>> a
(invalid Maxima object -- The maxima session in which this object was defined is␣
↪no longer running.)
>>> 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)                                                #␣
↪needs sage.libs.pari sage.symbolic
sage: a, b                                                                    #␣
↪needs sage.libs.pari sage.symbolic
(2, 3)
```

```
>>> from sage.all import *
>>> a = maxima(Integer(2)); b = gp(Integer(3))                                 ␣
↪               # needs sage.libs.pari sage.symbolic
>>> a, b                                                                     #␣
↪needs sage.libs.pari sage.symbolic
(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` – boolean (default: `False`); if `True`, display a message each time a process is sent a kill signal

EXAMPLES:

```
sage: gp.eval('a=10')                                                         #␣
↪needs sage.libs.pari
'10'
sage: sage.interfaces.quit.kill_spawned_jobs(verbose=False)
```

(continues on next page)

(continued from previous page)

```
sage: sage.interfaces.quit.expect_quitall()
sage: gp.eval('a=10')                                                    #␣
↪needs sage.libs.pari
'10'
sage: sage.interfaces.quit.kill_spawned_jobs(verbose=True)               #␣
↪needs sage.libs.pari
Killing spawned job ...
```

```
>>> from sage.all import *
>>> gp.eval('a=10')                                                    #␣
↪needs sage.libs.pari
'10'
>>> sage.interfaces.quit.kill_spawned_jobs(verbose=False)
>>> sage.interfaces.quit.expect_quitall()
>>> gp.eval('a=10')                                                    #␣
↪needs sage.libs.pari
'10'
>>> sage.interfaces.quit.kill_spawned_jobs(verbose=True)               #␣
↪needs sage.libs.pari
Killing spawned job ...
```

After doing the above, we do the following to avoid confusion in other doctests:

```
sage: sage.interfaces.quit.expect_quitall()
```

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

```
>>> from sage.all import *
>>> from sage.interfaces.quit import sage_spawned_process_file
>>> len(sage_spawned_process_file()) > Integer(1)
True
```

CHAPTER

FIFTY

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: 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]
```

```
>>> from sage.all import *
>>> indata = tmp_filename()
>>> f = open(indata, "w")
>>> _ = f.write("17\n42\n")
>>> f.close()
>>> l = read_data(indata, ZZ); l
[17, 42]
>>> f = open(indata, "w")
>>> _ = f.write("1.234\n5.678\n")
>>> f.close()
>>> l = read_data(indata, RealField(Integer(17))); l
[1.234, 5.678]
```

CHAPTER

FIFTYONE

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

INDEX

Non-alphabetical

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z